第 5 章 做市策略


31min

5.1 做市的基本概念

在一些活跃标的资产的交易活动中，普通的散户类投资者可以轻松地通过提交市价单、直接买卖目标标的资产或者标的资产的衍生品及其相关资产来直接参与市场交易。这样的市场中因为存在较多的投资者，所以资产流动性好。投资者更有可能在自己理想的合理价位出价，并且很快就可以找到与之交易的对手方。通过频繁有效的交易活动，资产的合理价值也可以在市场交易中逐渐被发现，从而使标的资产的价值得到充分体现，同时增强了市场的有效性。一方面推动了标的资产的价格发现，另一方面促进了市场资金的有效分配，通过市场的有效性也直接吸引了投资者的有效参与，为市场注入活力。

但在一些不活跃的交易资产上，由于缺乏足够的交易者，投资者难以实现这些资产的公允价值交易，也难以找到合适的对手方。这降低了市场的有效性。此时，做市商的作用凸显，他们为市场提供合适的交易活动，增加市场流动性。

5.2 高频做市策略

在高频交易策略中，做市策略，又称为市场制造策略，是一种提高市场流动性的关键策略类型。它指的是做市商(或流动性提供者)同时发布买入和卖出报价，以从市场中的买卖价差中获得利润。这种策略在市场中非常重要，既可以由那些被交易所指定为做市商并具有做市义务的机构执行，也可以由其他机构自愿提供流动性来获利。一般来讲，大多数低延迟交易机构没有承担做市义务，本书中主要讨论这一类机构的情况。

做市策略与低延迟交易策略的关系如图 5-1 所示。描述了低延迟交易策略和做市策略的密切关联。图中②部分指的是没有做市义务的低延迟高频做市策略。市场上绝大多数低延迟交易机构没有做市义务，本书的讨论主要基于这个细分类别。

做市商通过同时双向报价，利用成交价格在买卖价差间小幅高频波动来获利。低延迟做市策略的盈利来源是价格小波动的高频率变化，因此必须快速挂单以跟踪市场价格变动。

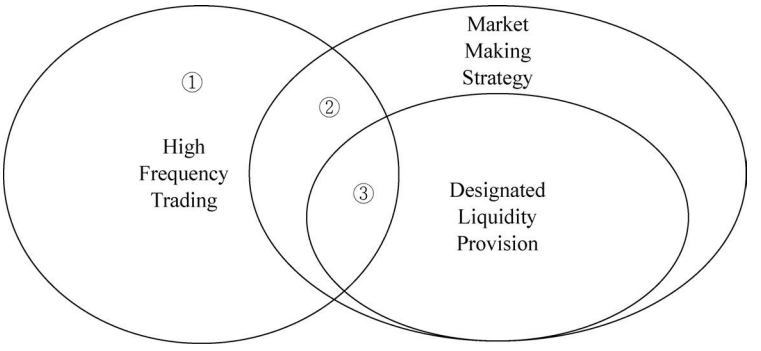

图 5-1 做市策略与低延迟交易策略的关系(图片来源 High-Frequency Trading)

尽管做市商希望他们的双向报价都能立刻成交,但并不总是如此,因为价格匹配需要时间。在市场行情出现明显大幅单边波动时,做市策略很可能由于逆向选择(Adverse Selection)只有与市场趋势相反的报单更有可能成交,从而导致单边净头寸积累而造成巨大的库存风险,从而产生大幅度亏损。

所以,一个优秀的做市策略的核心在于如何解决库存风险,如何根据标的资产的市场状态(流动性、波动率等)进一步地确定最优的买卖报单价格。

5.3 做市策略的收益来源

这种买卖价差是如何形成的?根据 Harold Demsetz 在 1968 年的有关纽约股市交易成本的研究,做市商买卖报价差的形成机制被首次提出并定义:投资者对资产供求的不平衡会导致价差产生,"买卖报价价差是有组织的市场为交易的即时性(immediacy)支付的加成"。做市策略通常在双边报价,通过成交价格在价差间的小幅高频波动来获利,而这里的幅度一般只有交易所最小出价价差的 1～2 倍,而不是单一方向大的趋势性变化。根据市场有效理论,股票价格在市场有效的状态下呈现为"随机漫步",价格的走势有着不可预测的性质,然而长期跟踪研究发现,价格的长期走势具有"均值回归(Mean Reversion)"的特点。均值回归在理论上具有必然性,价格走势不可能只升不降或者只降不升,价格保持正收益率或负收益率称为均值回避(Mean-aversion)。在均值回归理论中,均值回避的现象是暂时的,均值回归是必然的。资产价格偏离其内在价值的程度影响均值回归周期的长短。Chakraborty 和 Kearn(2011)通过推导理论和公式,进一步阐明了做市策略的绝对收益。假设所有的市场事件都在离散的时间点位 0,1,2,…,时刻 T 发生,并且在收盘时刻 T,做市策略必须平掉所有的单方向净头寸。研究表明做市策略的理论收益为 $\frac{1}{2}(K-Z^2)$,其中 $K=\sum_{t=1}^{T}|P_{t+1}-P_t|$ 表示价格波动的绝对幅度,$Z=P_T-P_0$ 表示收盘后平掉净头寸产生的净盈亏。此研究也进一步证明在均值回归的条件下,该理论收益的期望为正,即在均值回归的

假设下，做市策略确实可以产生绝对收益。根据这个理论收益公式，对于做市商来讲，一方面捕捉价格的窄幅波动非常重要，另一方面采取清仓操作以减少库存风险会对其绝对收益产生一定的减少效果。

5.4　经典做市策略 AS 模型

Avellaneda 和 Stoikov(2008)在引入库存风险考量的基础上建立了高频做市的 AS 模型。AS 模型的理论基础源于 Ho 和 Stoll(1980；1981)这两篇文章的研究结论，前者分析了在竞争环境中，做市商的报价与所有代理商的无差别报价相关，而后者则研究了一个做市商在考虑了存货风险的前提下，单项标的资产报价中的最优决策，即在资产的“真实价格”两侧创建最优买卖单。AS 模型在此基础上，研究了市场中单个做市商的最优决策行为，并用市场中间价代表所谓的“真实价格”。模型的建立主要分为两个步骤：首先，做市商在给定库存和风险偏好下，计算出自身对资产的无差异估值，即中间价格；其次，根据报价单与中间价之间的距离推算报价单被执行的概率，在此基础上结合市场环境和做市商的风险承受能力建立效用函数，推导出做市商的最优报价。

5.4.1　模型推导

1. 中间价格

做市商对于标的资产的无差异估值由下式给出：

$$\mathrm{d}S_u = \sigma \times \mathrm{d}W_u \tag{5-1}$$

中间价格的初始值 $S_t = s$，上式中 W_u 表示一维标准布朗运动。

2. 效用函数

做市商的目标是为了在时间 T 实现收益最大化，为了研究做市商的效用函数，Avellaneda 和 Stoikov 首先以不活跃的交易者为例考察了做市商的效用函数。不活跃的交易者指的是尚未提交报价任何报价单，在投资期间标的资产上有一定持仓的投资者。假设该交易者原来持有现金作为其库存时，该投资者的效用函数使用凸函数度量风险，此时交易者的效用函数如下式所示。

$$v(x,s,q,t) = E\left[-\exp(-\gamma(x+q\times S_T))\right] \tag{5-2}$$

$$v(x,s,q,t) = -\exp(-\gamma(x+q\times s))\times\exp((\gamma^2\times q^2\times\sigma^2(T-t))/2) \tag{5-3}$$

上述公式中变量的含义见 5.4.2 节。对于不活跃的交易者，当以 r^b 的价格买入一单位标的资产，成交后该交易者持有现金 $x-r^b$，库存增加一单位，若此交易行为对该交易者的效用不产生影响，则 r^b 表示该交易者的无差异买价(Reservation Bid Price)，即 r^b 应当满足：

$$v(x-r^b,s,q+1,t) = v(x,s,q,t) \tag{5-4}$$

通过同样的方式也可以建立无差异卖价(Reservation Ask Price)r^a 的等量关系式，解得

$$r^a(s,q,t)=s+(1-2q)\frac{\gamma\sigma^2(T-t)}{2}$$

$$r^b(s,q,t)=s+(-1-2q)\frac{\gamma\sigma^2(T-t)}{2} \tag{5-5}$$

$$r(s,q,t)=s-q\gamma\sigma^2(T-t)$$

其中，$r(s,q,t)$表示买卖价的均价。

上述讨论是针对有限的投资期间 $T-t$ 展开的，若从无限的时间长度来讨论，则该投资者的效用函数为

$$v(x,s,q)=E\left[\int_0^{\infty}v(x,s,q,t)\mathrm{d}t\right]$$

$$\omega=\frac{\gamma^2q^2\sigma^2}{2} \tag{5-6}$$

$$\Rightarrow v(x,s,q)=E\left[\int_0^{\infty}-\exp(-\omega t)\mathrm{d}t\right]$$

其中，ω 将决定投资者允许持有库存量的上界，一般设定为

$$\omega=\frac{1}{2}\gamma^2\sigma^2(q_{\max}+1)^2 \tag{5-7}$$

3. 构建限价单

为了解决最优报价决策问题，模型进一步研究了可以通过限价单交易参与市场投资的做市商的行为。

1）限价单报价及执行数量

做市商在中间价两端分别以 p^a 和 p^b 的价格报单，假设做市商可以连续无成本报价，报价单与中间价之间的距离 $\delta^a=p^a-s$、$\delta^b=s-p^b$，以及当前限价单的结构决定了该做市商限价单被执行的优先顺序。具体来讲，以限价买单为例，若市价卖单数量为 Q，当这批卖单最深的价位 p^Q 低于做市商限价买单报价 p^b 时，限价单被击穿成交，而实证研究表明市价卖单最深价位与中间价的差价 ΔP 与市价卖单数量的对数值成正比，公式如下：

$$\Delta p=p^Q-s\propto\ln(Q) \tag{5-8}$$

经过时间 t 后，做市商分别持有 N_t^a 手空单，N_t^b 手多单。根据研究，假设 N_t^a、N_t^b 分别服从速率为 λ^a 和 λ^b 的泊松过程，λ^a 和 λ^b 表示限价单分别被市价单击穿的概率，当 δ 超出 ΔP 时，限价单将不会被击穿，从而得到 $\lambda(\delta)=A\exp(-\kappa\delta)$。

2）最优化问题

经过时间 t 后，做市商持有现金 X_t，满足：

$$\mathrm{d}X_t=p^a\mathrm{d}N_t^a-p^b\mathrm{d}N_t^b \tag{5-9}$$

净库存为 $q_t=N_t^b-N_t^a$。此时做市商面临的最优化问题是：

$$u(x,s,q,t)=\max_{\delta^a,\delta^b}E[-\exp(-\gamma(X_T+q_TS_T))] \tag{5-10}$$

上述等式也需要同时满足：

$$u(x,s,q,t)=u(x-r^b,s,q+1,t)=u(x+r^a,s,q-1,t) \tag{5-11}$$

可以通过求解上述价值函数的 Hamilton-Jacobi-Bellman(HJB)方程解得 r^a 和 r^b 的均值即中间价及 δ^a 和 δ^b 的和：

$$r(s,q,t)=\frac{r^a+r^b}{2}=s-q\gamma\sigma^2(T-t)$$

$$\delta^a+\delta^b=\frac{2}{\gamma}\ln(1+\gamma/\kappa) \tag{5-12}$$

3）带库存惩罚项(ASQ)

根据以上假设建模，经典的 Avellaneda Stoikov 做市模型可以根据自身的净头寸计算出预定价格，然后根据市场的成交概率，围绕这个预定价格做市商得到最优的买卖报价。这个模型虽然能够较好地模拟市价单的成交情况，增加了库存惩罚项，但面对极端行情仍然存在巨大的库存风险，因此为了更好地优化库存管理，Olivier 和 Charles 等提出了根据做市商风险厌恶程度，增加库存的最大值限 $Q_{\max}$。在库存达到最值时，即停止向增加库存方向报价，而只做反方向报价，从而限制库存的进一步增加。

此时，建立目标函数，做市商期望在基准时间 T 内使 PnL 最大化。使用常数绝对风险厌恶效用函数(CARA)进行优化：

$$\sup_{(\delta_t^a)_t,(\delta_t^b)\in A} E[-\exp(-\gamma(X_T+q_TS_T))] \tag{5-13}$$

这里 A 是上面定义的可预测过程的集合，γ 是量化做市商风险厌恶程度的绝对风险规避系数，X_T 是在时间 T 处的现金量，q_T、S_T 是在时间 T 的标的资产库存价值。

在求解以上优化问题中，引入 HJB 方程，可以通过求解由以下方程组成的偏微分方程组：

当 $|q|<Q_{\max}$ 时：

$$\partial_t u(t,x,q,s)+\frac{1}{2}\sigma^2\partial_{ss}^2u(t,x,q,s)+\sup_{\delta^b}\lambda^b(\delta^b)[u(t,x-s+\delta^b,q+1,s)-u(t,x,q,s)]+$$

$$\sup_{\delta^a}\lambda^a(\delta^a)[u(t,x+s+\delta^a,q-1,s)-u(t,x,q,s)]=0 \tag{5-14}$$

当 $|q|=Q_{\max}$ 时：

$$\partial_t u(t,x,Q,s)+\frac{1}{2}\sigma^2\partial_{ss}^2u(t,x,q,Q,s)+\partial_t u(t,x,Q,s)+\frac{1}{2}\sigma^2\partial_{ss}^2u(t,x,q,Q,s)+$$

$$\sup_{\lambda^b}(\delta^b)\ [u(t,x-s+\delta^b,-Q+1,s)-u(t,x,-Q,s)]=0 \tag{5-15}$$

当 $|q|=-Q_{\max}$ 时：

$$\partial_t u(t,x,Q,s)+\frac{1}{2}\sigma^2\partial_{ss}^2u(t,x,q,Q,s)+\partial_t u(t,x,Q,s)+\frac{1}{2}\sigma^2\partial_{ss}^2u(t,x,q,Q,s)+$$

$$\sup_{\lambda^b}(\delta^b)\ [u(t,x-s+\delta^b,-Q+1,s)-u(t,x,-Q,s)]=0 \tag{5-16}$$

限制条件为

$$q\in\{-Q,\cdots,Q\}\text{for}(t,s,x)\in[0,T]\times R^2$$

$$\forall_q \in \{-Q,\cdots,Q\}, u(T,x,q,s) = -\exp(-\gamma(x+qs)) \tag{5-17}$$

可以得到

$$\delta_t^{b*} \simeq \frac{1}{\gamma}\ln\left(1+\frac{\gamma}{k}\right)+\frac{1+2q}{2}\gamma\sigma^2(T-t) \tag{5-18}$$

$$\delta_t^{a*} \simeq \frac{1}{\gamma}\ln\left(1+\frac{\gamma}{k}\right)+\frac{1-2q}{2}\gamma\sigma^2(T-t) \tag{5-19}$$

做市商围绕参考价格进行最优报价,其中买单报价为 $r-\delta_b$,卖单报价为 $r+\delta_a$。这里会存在几种极端情况,例如当买单最优报价高于市场中标的资产的挂单卖一价时(卖单最优报价同理),做市商可能就会直接使用市价单进行平仓交易。另外,由于 AS 模型中累计库存惩罚项只体现在基准价格 r 中,为优化库存管理,当 $|q|=Q_{\max}$ 时,做市商将不再向库存增加方向报单。

4) 结合趋势项(漂移项)

上面第一部分介绍了经典的 AS 做市模型,并添加了与做市商风险偏好相关的库存优化,然而,我们在研究过程中对参考价格运动服从布朗运动的假设与实际是存在较大偏差的,很多时候,标的资产的市场价格运动时存在一定的趋势,即应有

$$\mathrm{d}S_t = \mu d_t + \sigma \mathrm{d}W_t \tag{5-20}$$

其中,μ 为价格运动中的趋势强度。与经典 AS 做市模型的最优解解法相似,我们将 S_t 过程假设替换,可以得到做市商的近似最优解为

$$\delta_{\infty}^{b*}(q) \simeq \frac{1}{\gamma}\ln\left(1+\frac{\gamma}{k}\right)+\left[-\frac{\mu}{\gamma\sigma^2}+\frac{2q+1}{2}\right]\sqrt{\frac{\sigma^2\gamma}{2kA}\left(1+\frac{\gamma}{k}\right)^{1+\frac{k}{\gamma}}} \tag{5-21}$$

$$\delta_{\infty}^{a*}(q) \simeq \frac{1}{\gamma}\ln\left(1+\frac{\gamma}{k}\right)+\left[\frac{\mu}{\gamma\sigma^2}-\frac{2q-1}{2}\right]\sqrt{\frac{\sigma^2\gamma}{2kA}\left(1+\frac{\gamma}{k}\right)^{1+\frac{k}{\gamma}}} \tag{5-22}$$

5.4.2 AS 模型通俗解读与应用

AS 是经典的做市模型,主要是双向报价的同时控制库存风险。一般来讲在震荡行情中,特别是在波动率不高的情况下适合做市,通过 AS 的经典论文得到两个关键公式,下面来解读式(5-23)和式(5-24)的含义。

$$r(s,q,t) = \frac{r^a + r^b}{2} = s - q\gamma\sigma^2(T-t) \tag{5-23}$$

$$\delta^a + \delta^b = \gamma\sigma^2(T-t) + \frac{2}{\gamma}\ln(1+\gamma/\kappa) \tag{5-24}$$

上述公式的参数含义如下:

$r(s,q,t)$即 Reservation Price,翻译为预定价格或无差异价格。它是做市商愿意买入或持有该资产的最高价格,也是做市商愿意卖出该资产的最低价格。它反映了做市商对资产价值的评估。

s=Current Market Mid Price(中间价,也就是(最佳卖价+最佳买价)/2)

q=Quantity of Assets in Inventory of Base Asset(做市商持有多少股票数量)

σ=Market Volatility(市场波动率,可以用 std 标准差来表示)

T=Closing Time(测量周期何时结束(标准化为 1))

t=Current Time(T 被标准化为 1,因此 t 是一个分数)

t 是当前时间,T 是结束时间,如果投资标的是 24h 连续交易的品种,则 T 可以被设置为无穷大。如果操作品种是非连续交易的品种,例如商品期货,则需要调整 T 的设置以满足日内交易的设定,例如当日收盘前 5min 清仓离场。

按照通俗的理解就是,AS 策略主要解决两个核心问题。

(1) 库存风险(特别是单边行情,如持有较多方向不利的仓位,造成较大损失)。

(2) 找到最优的买入价格、卖出价格。

δ^a,δ^b=bid/ask spread,并有对称性,所以 $\delta^a=\delta^b$;也就是一段时间内,bid/ask 上蹿下跳的幅度计算,大部分报价是按照 Reservation Price$\pm\delta$ 进行对称性报价的,如果 $\delta^a=\delta^b$,实际上做市也就是一种网格交易。在中间价格之下的一定距离设置买单,在中间价之上的一定距离设置卖单,如果买卖单都成交了,则总持仓不变,总盈利为 $\delta^a+\delta^b$(含手续费)。相当于低买高卖。

γ=Inventory Risk Aversion Parameter,翻译过来就是规避库存风险的参数,当取值很大时,预定价格就和中间价差距远。

κ=Order Book Liquidity Parameter,是评估单本订单密度参数。也就是 κ 值越大,参与买卖的人比较多,出价也比较均衡,订单量也比较大。

如果 κ 值很小,则意味着最佳买价(Best Buy)和最佳卖价(Best Sell)挂单比较小。如果市场上出现大额市价单,直接就可能打穿最佳买价和最佳卖价挂单并推动中间价移动。

AS 策略就是围绕 Reservation Price 进行报价的,例如卖价 Reservation Price+δ;买入价 Reservation Price−δ。采用这样报价的方法的弊端就是,如果发生单边下跌行情,你可能就满仓持有资产,产生较大亏损。如果单边上涨,你就无货可卖,俗称卖飞了。

股指期货 IF 震荡下跌图如图 5-2 所示,如果采用类似固定值网格操盘法,交易结果就是持有的 IF 多头仓位不断增加,而且 IF 的点位不断下降导致亏损。

AS 通过 3 方面的因子来规避持仓增加风险:

(1) 设定目标持仓量 $q_{\max}$,也就是持有股票数量最大值。

也就是查询当前持仓和目标仓位的差值。例如你有 100 万元人民币,做 IF 做市,可以设定你的合理持仓是 4 手 IF 合约和 50 万元人民币(合约和现金各半,设定 1 手 IF 约 12.5 万元)。策略初始化持有 100 万元人民币和 0 手合约,所以仓位差值 $q=0-4=-4$,初始化的时候,策略需要尽力去买入 IF 合约,而当持仓 5 手 IF 合约,那么按照 AS 策略,就要平仓 1 手 IF 合约,维持持仓 4 手持仓目标;报价策略如下:

当前持仓小于 $q_{\max}$ 值,就要提高预定价格,买单执行概率增大,卖单执行概率减少。

当前持仓大于 $q_{\max}$ 值,就要降低预定价格,卖单执行概率增大,买单执行概率减少。

图 5-2　股指期货 IF 震荡下跌图

(2) 持仓风险 γ。

设置 γ 越大,公式后面部分乘积就越大,数字和中间价偏离就大。如果设置得很小就很靠近中间价。当 γ 被设置为 0 时,预定价格 r 非常接近现在的中间价,这就是固定网格值的网格策略,和网格策略一样的收益和风险承担。

交易持续的时间($T-t$),在结束时间 T 目标是最大化其损益的预期指数效用。读者可能已经注意到,笔者没有在主要因素列表中添加波动率(σ),尽管它是公式的一部分。这是因为波动值取决于市场价格的变动,而不是做市商定义的因素。如果市场波动性增加,则预定价格与市场中间价之间的距离也将增加。

模型公式的第二部分是关于寻找做市商订单在订单簿上的最佳位置,以提高盈利能力。

订单簿流动性/密度是如何计算 κ 的,相关论文中有很多数学细节,解释了他们是如何通过假设指数到达率得出这个因子的,但就目前而言,重要的是要知道,使用较大的 κ 值,是假设订单更加密集,并且最优价差必须更小,因为市场竞争更加激烈(也就是挂单和中间价的偏离度就很小)。另外,使用较小的 κ,假设订单的流动性较低,可以使用的价差就变大(也就是挂单和中间价的偏离度就很大)。结合预定价格和最优价差可以计算出挂单价格,这就是 AS 模型魔术发生的地方。

AS 模型的执行逻辑非常简单。

Step1:根据目标库存计算预订价格。

Step2:计算最优买卖价差。

Step3:使用预定价格作为参考创建市场订单。

Step4:bid_price=预定价格-最优价差/2。

Step5:ask_price=预定价格+最优价差/2。

价格变动如图 5-3 所示。

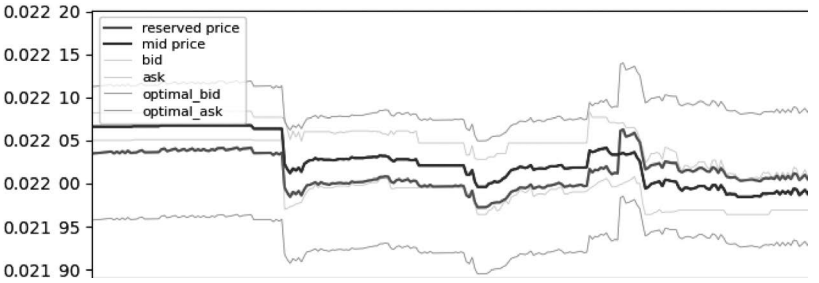

图 5-3　价格变动图

如何动态地计算预定价格，图 5-3 给出了一个直观的概念，也就是前半段预定价格小于中间价，因为做市商是有库存要抛出来的，所以让 ask 的价格贴近中间价，这样就可以增加 ask 订单的成交，从而使做市商持有的股票更容易抛出。后半段，做市商手里存货减少，需要进行补仓操作，所以让预定价格提高，让预定价格 bid price 更接近中间价，这样可以增加买进的概率，从而快速地补仓。

(3) 计算输入的参数。

回顾本节，提到过 AS 模型用于计算预定价格和最优价差的 3 个主要因素。

(1) 库存状况(q)：做市商持有的目标仓位。

(2) 交易时段结束前的时间($T-t$)：是交易时段结束前剩余的时间。AS 模型被创建用于传统金融市场，在传统金融市场中，交易有开始时间和结束时间。这个参数背后的原因是，随着交易日接近尾声，做市商希望拥有与交易日开始时相似的库存头寸，因此，随着交易日接近尾声，订单价差将更小，而预定价格在重新平衡库存方面将更加“激进”。

(3) 风险因子(γ)和深度因子(κ)比较复杂，将在 5.4.3 节介绍。

5.4.3　AS 模型工程化实现

再次回顾公式，这次对于这些公式里面一些希腊字母的取值是如何计算的，如何做工程化并应用到生产环境中进行一些说明。

$$r(s,q,t)=\frac{r^a+r^b}{2}=s-q\gamma\sigma^2(T-t)$$

$$\delta^a+\delta^b=\gamma\sigma^2(T-t)+\frac{2}{\gamma}\ln(1+\gamma/\kappa) \tag{5-25}$$

公式里面有一些重要参数，直观取值的有以下两个。

s=Current Market Mid Price(中间价，也就是(最佳卖价+最佳买价)/2)。

q=Quantity of Assets in Inventory of Base Asset(做市商持有多少股票数量)。

当然 $T-t$ 也是可以直接定义的，但是对于连续合约，7×24h 交易的数字货币市场，如何来定义 T，在计算中如何处理是需要一些技巧的，采用的方法是对 T 进行归一化。

需要计算的值有以下几个。

σ=Market Volatility(市场波动率，可以用 std 标准差来表示)。

γ=Inventory Risk Aversion Parameter(规避库存风险的参数)。

κ=Order Book Liquidity Parameter(是一个评估订单簿的订单密度参数。也就是κ值越大,参与买卖的人越多,出价也比较均衡,流动性较好)。

$\delta^a+\delta^b$ 称为最优价差,$\delta^a=\delta^b$,也就是 Bid Spread 和 Ask Spread 是相等的。

所以 AS 模型做决策的价格就是:

$$\text{Bid Price}=\text{预定价格}-\text{最优价差}/2$$

$$=s-q\gamma\sigma^2(T-t)-0.5\gamma\sigma^2(T-t)-\frac{1}{\gamma}\ln\left(1+\frac{\gamma}{\kappa}\right) \tag{5-26}$$

$$\text{Ask Price}=\text{预定价格}+\text{最优价差}/2$$

$$=s-q\gamma\sigma^2(T-t)+0.5\gamma\sigma^2(T-t)+\frac{1}{\gamma}\ln\left(1+\frac{\gamma}{\kappa}\right) \tag{5-27}$$

1. 计算 γ

假设用户在配置策略时设置了 Min Spread 和 Max Spread 参数。为了计算最大可能风险因子(γ),作者使用初始价差最优买入/卖出中间价($\delta^a+\delta^b$),其不应小于最小价差,也不应大于最大价差(与中间价相关)。

由于保留价和中间价之间的差值(Δ)和最优价差是($T-t$)的函数,为了降低 q 的绝对值(接近做市商底仓),可以肯定地说,最优出价和要求中间价的价差将随着时间的推移而减少,因此,以下计算将以 $t=0$ 的时刻为中心,其价差是最宽的(也就是启动策略时,bid/ask 报价和中间价偏离最大,以便于成交和持仓,临近 T 时刻,要清理库存,采用这种方式动态地控制仓位)。

库存过剩时的价格水平和价差分布如图 5-4 所示,因此预定价格低于中间价格,价差会相应地调整。

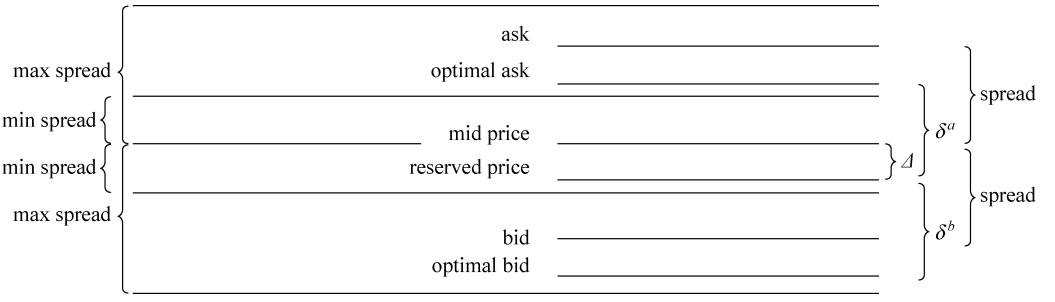

图 5-4 库存过剩时的价格水平和价差分布

$q>0$(Inventory Needed to be Decreased),持仓超过预期,也就是需要开始减仓:

$$\text{Spread\_optimal\_ask}\leqslant\text{Spread optimal\_bid Spread}$$

要符合最大和最小价差:也就是 Ask 的报价更低一些,Bid 的报价更远离 Reserve Price,这样可以卖掉一些库存,购入库存的概率更低一些。

$$\text{Spread\_optimal\_ask}_{t=0}\geqslant\text{Min Spread}$$

$$\text{Spread\_optimal\_bid}_{t=0}\leqslant\text{Max Spread}$$

现在计算 $t=0$ 时这些价差的表达式。首先计算 ask 价差：

$$\text{Spread\_optimal\_ask}_{t=0}=\text{Optimal\_ask}_{t=0}-s$$

$$\text{Optimal\_ask}_{t=0}=r(s,0)+\frac{(\delta^{a}_{t=0}+\delta^{b}_{t=0})}{2} \tag{5-28}$$

把相关的值代入有

$$\text{Spread\_optimal\_ask}_{t=0}=s-\left[s+q\gamma\sigma^{2}+\frac{1}{2}\left[\gamma\sigma^{2}+\ln\left(1+\frac{\gamma}{\kappa}\right)\right]\right]\geqslant \text{Min Spread} \tag{5-29}$$

对上述公式进行整理，因此，得出的表达式为

$$\left(\frac{1}{2}-q\right)\gamma\sigma^{2}+\frac{1}{2}\ln\left(1+\frac{\gamma}{\kappa}\right)\geqslant \text{Min Spread} \tag{5-30}$$

同样，对于 Bid Spread 是用户设置的最大价差：

$$\text{Spread\_optimal\_bid}_{t=0}=s-\text{optimal bid } t_{t=0} \tag{5-31}$$

$$\text{Spread\_optimal\_bid}_{t=0}=s-\left[s-q\gamma\sigma^{2}-\frac{1}{2}\left(\gamma\sigma^{2}+\ln\left(1+\frac{\gamma}{\kappa}\right)\right)\right]\leqslant \text{Max Spread} \tag{5-32}$$

最后化简整理得到表达式：

$$\left(\frac{1}{2}+q\right)\gamma\sigma^{2}+\frac{1}{2}\ln\left(1+\frac{\gamma}{\kappa}\right)\leqslant \text{Max Spread} \tag{5-33}$$

将两个不等式相加，得出以下结果（可以观察统计 Min Spread 和 Max Spread 进行反推 γ 的值，因为 γ 和 q 值是不变的）：

$$\left(\frac{1}{2}-q\right)\gamma\sigma^{2}+\frac{1}{2}\ln\left(1+\frac{\gamma}{\kappa}\right)\geqslant \text{Min Spread} \tag{5-34}$$

$$-\left(\frac{1}{2}+q\right)\gamma\sigma^{2}-\frac{1}{2}\ln\left(1+\frac{\gamma}{\kappa}\right)\geqslant -\text{Max Spread} \tag{5-35}$$

将式(5-34)和式(5-35)合并化简，得出的表达式为

$$2q\gamma\sigma^{2}\leqslant \text{Max Spread}-\text{Min Spread} \tag{5-36}$$

$$q<0(\text{Inventory Needed to be Increased})$$

同样，对于相反的情况，如果 $q<0$，则得出的最终表达式为（可以观察统计 Min Spread 和 Max Spread 进行反推 γ 值，因为 γ 和 q 值是不变的，这段时间段是需要增加库存的情况）

$$-2q\gamma\sigma^{2}\leqslant \text{Max Spread}-\text{Min Spread} \tag{5-37}$$

由于风险因子 γ 为非负值，通过计算此最大阈值，我们现在有了所有可能的 γ 值的范围。后面提到 Inventory Risk Aversion(IRA)是一个从 0 到 1 的系数，控制仓位的按钮，它将约束 γ 的取值范围。观察统计 Min Spread 和 Max Spread，这样分子不变，然后查看 σ 的最小波动，以及最小持仓的时候，这时 γ 的值就最大了，同理，当波动率最大，库存最大，γ 就取得最小值，然后 $\gamma_{\max}$ 乘以 IRA 系数，这个系数就可以通过 bid ask 发出的报价来调整库存的系数。

最终方程式 γ：

$$\gamma=\gamma_{\max}\times \mathrm{IRA}=\frac{\text{Max Spread}-\text{Min Spread}}{2\parallel q\parallel \sigma^2}\times \mathrm{IRA} \tag{5-38}$$

2. 计算 $\kappa((\delta_a+\delta_b)_{\max})$

将选择订单簿深度因子(κ),以便算法从 $t=0$ 时的最大可能价差开始。这个决定似乎是任意的,但其背后的论点是通过更大范围的价差实现策略盈利能力的最大化,因此,从计算开始,首先确定 $t=0$ 时的最大波动可能。

$$\text{spread}_{t=0}=\frac{\delta_a+\delta_b}{2}\pm\Delta \tag{5-39}$$

最深度的订单是远离中间价加上 Δ,按照如下公式进行计算:

$$\begin{aligned}
&\frac{\delta_a+\delta_b}{2}+\Delta\leqslant \text{Max Spread}\\
&(\delta_a+\delta_b)_{\max}=2\text{Max Spread}-2\Delta\\
&\qquad=2\text{Max Spread}-2\parallel q\parallel\gamma\sigma^2\\
&\qquad=2\text{Max Spread}-2\parallel q\parallel\sigma^2\times\frac{\text{Max Spread}-\text{Min Spread}}{2\parallel q\parallel\sigma^2}\times\mathrm{IRA}\\
&\qquad=(2-\mathrm{IRA})\times\text{Max Spread}+\mathrm{IRA}\times\text{Min Spread}
\end{aligned} \tag{5-40}$$

现在,从 $t=0$ 时的最大最优价差来看,κ 可以用本书的公式推断为该 Spread 的函数:

$$\begin{aligned}
(\delta_a+\delta_b)_{t=0}&=(2-\mathrm{IRA})\times\text{Max Spread}+\mathrm{IRA}\times\text{Min Spread}\ \kappa((\delta_a+\delta_b)_{t=0})\\
&=\frac{\gamma}{\exp\left\{\dfrac{(\delta_a+\delta_b)_{t=0}\gamma-\sigma^2\gamma^2}{2}\right\}-1}
\end{aligned} \tag{5-41}$$

3. 计算η

还记得 Order Amount Shape Factor(η)是订单金额的形参,借用 2018 年 Fushimi 的论文

$$\begin{aligned}
\phi_t^{\text{bid}}&=\begin{cases}\phi_t^{\max} & q_t<0\\ \phi_t^{\max}\times \mathrm{e}^{-\eta q t} & q_t>0\end{cases}\\
\phi_t^{\text{ask}}&=\begin{cases}\phi_t^{\max} & q_t<0\\ \phi_t^{\max}\times \mathrm{e}^{-\eta q t} & q_t>0\end{cases}
\end{aligned} \tag{5-42}$$

$\phi_t^{\text{bid}},\phi_t^{\text{ask}}$ 是 t 时刻的 bid,ask 的订单大小。$\phi_t^{\max}$ 是 t 时刻的 bid,ask 的订单最大值。η 是一个形参。基本上,从策略提案中提交的两份订单来看,不利于达到目标库存的订单将根据策略距离目标 q 的距离呈指数级减少。

利用前面定义的 Inventory Risk Aversion(IRA)参数,指数衰减函数中的衰减量将由 IRA 控制:

$$q_{\text{decay}}=\frac{\text{Totalinventoryinbase\_asset}}{\mathrm{IRA}}\eta=\frac{1}{q_{\text{decay}}} \tag{5-43}$$

如果 IRA→0⇒γ→0，则会发生什么？当 IRA→0 也意味着 γ→0，IRA 是设置库存规避的旋钮，当 IRA→0 时，这个旋钮就会失去功能，整个报价变成无库存风险规避。也就是用户的报价就是围绕 Mid Price 进行对称性报价，想象一下 Spread 的值是什么？做一下数学计算。

$$\text{如果}\quad \text{IRA}\to 0\Rightarrow\gamma\to 0 t=0\Rightarrow(T-t)=1 \tag{5-44}$$

$$\lim_{\gamma\to 0} r(s,q,t=0,\sigma)=s \tag{5-45}$$

$$\lim_{\gamma\to 0}\delta^{a}+\delta^{b}(q,t=0,\sigma)=\lim_{\gamma\to 0}\frac{2}{\gamma}\ln\left(1+\frac{\gamma}{\kappa}\right)=\lim_{\gamma\to 0}\frac{2}{\gamma}\frac{\gamma}{\kappa}=\frac{2}{\kappa} \tag{5-46}$$

最后，计算 κ 值，这意味着，如果 IRA→0⇒γ→0 波动 Spread r=Mid Price，则这个值会被固定。

$$r=s \text{ 如果 } \gamma\to 0\Rightarrow \text{IRA}\to 0(\delta_a+\delta_b)=(\delta_a+\delta_b)_{\max}$$

$$=(2-\text{IRA})\times\text{Max Spread}+\text{IRA}\times\text{Min Spread}=2\times\text{Max Spread r}=s \tag{5-47}$$

因此，在 γ 为 0 的情况下，这与常规的纯做市策略相同，对称价差等于中间价附近的最大价差。这样，纯做市策略就成为 AS 做市策略的特例。关于 AS 策略的复现和回测，详见第 9 章的回测案例。考虑排队模型与成交概率等因素的精细回测的源码，可扫描目录上方二维码下载。

5.5　经典做市策略 GP 模型

5.5.1　马尔可夫链

1. 简介

马尔可夫链(Markov Chain)通常指一类在概率论和数理统计中具有马尔可夫性质且存在于离散的指数集和状态空间内的随机过程。适用于连续指数集的马尔可夫链被称为马尔可夫过程，也被视为连续时间马尔可夫链，与离散时间马尔可夫链相对应，因此马尔可夫链是一个较为宽泛的概念。

马尔可夫链的命名来自俄国数学家安德雷·马尔可夫，以纪念其首次提出马尔可夫链及其对马尔可夫链收敛等性质研究所做出的贡献。

2. 定义

1) 随机过程

将一族无穷多个、相互有关的随机变量叫作随机过程，例如

$$\{x_n,n=0,1,2,\cdots\}=x_0,x_1,x_2,\cdots \tag{5-48}$$

通常将随机过程记作 $X(t)$，随机过程 $X(t)$是一组依赖于实参数 t 的随机变量，t 一般具有时间的含义。根据 t 是否连续，随机过程又可分为离散时间随机过程和连续时间随机过程。

2) 马尔可夫性质

当一个随机过程在给定现在状态及所有过去状态的情况下，其未来状态的条件概率分

布仅依赖于当前状态；换句话说，在给定现在状态时，它与过去状态（该过程的历史路径）是条件独立的，那么此随机过程即具有马尔可夫性质，即

$$P(X_{n+1}=x_{n+1}\mid X_n=x_n,X_{n-1}=x_{n-1},\cdots,X_1=x_1,X_0=x_0)=P(X_{n+1}=x_{n+1}\mid X_n=x_n) \tag{5-49}$$

或者说：

$$(X_{n+k},n\geqslant 0)\overset{d}{=}(X_n,n\geqslant 0)\quad \text{如果 } X_0=X_k=x \tag{5-50}$$

3）马尔可夫链

马尔可夫链是一组具有马尔可夫性质的随机变量集合（通常是离散的）。具体地，对概率空间$(\Omega,\mathcal{F},\mathbb{P})$内以一维可数集为指数集的随机变量集合$X$，若随机变量的取值都在可数集$S$内，并且随机变量的条件概率满足马尔可夫性质，则$X$被称为马尔可夫链，可数集被称为状态空间，马尔可夫链在状态空间内的取值称为状态。

转移概率：

对于离散的情况，我们将马尔可夫过程中现在处于状态i，下一步转移至状态j的单步转移概率记为P_{ij}。

$$\boldsymbol{P}=\begin{bmatrix}P_{00} & P_{01} & P_{02} & \cdots\\ P_{10} & P_{11} & P_{12} & \cdots\\ \vdots & \vdots & \vdots & \vdots\\ P_{i0} & P_{i1} & P_{i2} & \cdots\end{bmatrix} \tag{5-51}$$

其中

$$P_{ij}\geqslant 0,\quad \forall i,j\in S$$

$$\sum_{j=0}^{\infty}P_{ij}=1,\quad i=0,1,2,\cdots$$

特征向量（以下公式描述的是在给定当前状态$X_n=x_n$的条件下，下一状态X_{n+1}转移到状态x的概率）：

$$P(X_{n+1}=x\mid X_n=x_n) \tag{5-52}$$

在马尔可夫链上找一种状态作为起点做随机漫步，做n步。之后找到所有状态的对应概率，形成一个概率分布。每种状态的概率就是把每种状态出现的次数除以总步数。

假设马尔可夫链有3种状态，n为10，最后得出了这样的分布：$\frac{4}{10},\frac{2}{10},\frac{4}{10}$。

可以用Python写一个模拟$n=100\,000$的随机漫步程序，会发现这些概率分布会被收敛到0.351 91，0.212 45，0.435 64。这种概率分布有一个特别的名字叫作稳态分布。

$$\boldsymbol{A}=\begin{bmatrix}0.2 & 0.6 & 0.2\\ 0.3 & 0 & 0.7\\ 0.5 & 0 & 0.5\end{bmatrix} \tag{5-53}$$

$$\boldsymbol{\pi}_0=[0\ 1\ 0]$$

把这个行向量与转置矩阵相乘，

$$\boldsymbol{\pi}_0\boldsymbol{A} = [0.3\ 0\ 0.7]$$

就可以得到状态 2 的未来概率，然后把这个结果替换到$\boldsymbol{\pi}_0$的位置，计算 $\boldsymbol{\pi}_1\boldsymbol{A}$。

$$\boldsymbol{\pi}_1\boldsymbol{A} = [0.3\ 0\ 0.7]\begin{bmatrix}0.2 & 0.6 & 0.2\\ 0.3 & 0 & 0.7\\ 0.5 & 0 & 0.5\end{bmatrix} \tag{5-54}$$

这样一直进行下去。

$$\lim_{n\to\infty}\prod_{i=n}^{0}\pi_i\prod_{i=1}^{n}\boldsymbol{A} \tag{5-55}$$

如果存在一个稳态，则在某个点后，输出的行向量应该与输入的行向量完全相同，我们用 $\boldsymbol{\pi}$ 来代表这个特殊的行向量。

$$\boldsymbol{\pi}\boldsymbol{A} = \boldsymbol{\pi}$$

$\boldsymbol{\pi}$ 其实是 $\boldsymbol{A}$ 的左特征向量。现在特征向量还需要满足另一个条件。$\boldsymbol{\pi}$ 的所有元素加起来必须等于 1。因为它代表的是概率分布，因此解完这两个等式之后，就得到了这样的稳态。

$$\boldsymbol{\pi}[1] + \boldsymbol{\pi}[2] + \boldsymbol{\pi}[3] = 1$$

$$\boldsymbol{\pi} = \begin{bmatrix}\frac{25}{71} & \frac{15}{71} & \frac{31}{71}\end{bmatrix}$$

实际上可以计算是否存在多个稳态，只需查看是否存在不止一个特征值等于 1 的特征向量。

状态的态和类：

如果状态 A 到状态 B 之间有箭头，那就说明状态 A 到状态 B 存在非零的转移概率。从任何特定状态出发的转移概率总和为 1。在从某种状态开始的随机漫步中，重新回到这种状态的概率小于 1。无法确切地知道会不会回到这里。在这种情况下，自身返回概率小于 1 的状态，称为“暂态”。

从某种状态开始随机漫步，只需一段时间，肯定能回到这种状态。在这种情况下，回到自身状态的概率是 1，称为“常返态”。有些状态无法从其他状态回到的情况，称这个马尔可夫链为可约的。每种状态都能从其他状态到达，或者说可以从任何一种状态到达其他状态的链，称为不可约链。感兴趣的读者可以去了解赌徒的毁灭马尔可夫链。一个马尔可夫链约出所有不可约链，每个不可约链称为一个通信类。

高阶转置矩阵与平衡状态：

从状态 i 到状态 j 刚好需要 n 步到达的概率是多少？

$$P_{ij}(n) = A_{ij}^{n} \tag{5-56}$$

例如

$$\boldsymbol{A} = \begin{bmatrix}0.5 & 0.2 & 0.3\\ 0.6 & 0.2 & 0.2\\ 0.1 & 0.8 & 0.1\end{bmatrix} \tag{5-57}$$

$$P_{02}(2)$$

下面的公式中每个元素代表了一个从 i 到 j 的转移概率：

$$A_{01}\times A_{12}+A_{00}\times A_{02}+A_{02}\times A_{22} \tag{5-58}$$

调整一下这些项：

$$A_{00}\times A_{02}+A_{01}\times A_{12}+A_{02}\times A_{22} \tag{5-59}$$

这样就变成了两个向量的乘积了：

$$[A_{00}\ A_{01}\ A_{02}]\times\begin{bmatrix}A_{02}\\A_{12}\\A_{22}\end{bmatrix} \tag{5-60}$$

这类似于矩阵乘法中一个矩阵内元素的计算方式。这就能推广到 n 步了，也就是 n 阶转置矩阵。

这里其实用到了 Chapman-Kolmogrov 定理，之所以能使用它，是因为马尔可夫性质。

$$P_{ij}(n)=\sum_k P_{ik}\times P_{kj}(n-r) \tag{5-61}$$

前面计算从状态 0 到状态 2 的概率时，就使用了这种方法：

$$P_{02}(2)=P_{00}(1)\times P_{02}(1)+P_{01}(1)\times P_{12}(1)+P_{02}(1)\times P_{22}(1)$$

这里的 r 是 1。

定理证明：

$$\begin{aligned}P_{ij}(n)&=P(X_n=j\mid X_0=i)\\&=\sum_k P(X_n=j,X_r=k\mid X_0=i)\\&=\sum_k\frac{P(X_n=j,X_r=k,X_0=i)}{P(X_0=i)}\\&=\sum_k\frac{P(X_n=j,X_r=k,X_0=i)\times P(X_r=k,X_0=i)}{P(X_0=i)}\\&=\sum_k\frac{P_{kj}(n-r)\times P(X_r=k,X_0=i)}{P(X_0=i)}\\&=\sum_k P_{ik}\times P_{kj}(n-r)\end{aligned} \tag{5-62}$$

简单来讲，稳态分布是长期访问每种状态的概率。除了以上讲的如何从特征向量来找到它，还可以从另一个角度来看到它们。长期来看，意味着无数次的转移。

$$\lim_{n\to\infty}\boldsymbol{A}^n$$

$$\boldsymbol{A}^{\infty}=\begin{bmatrix}0.4444 & 0.3333 & 0.2222\\0.4444 & 0.3333 & 0.2222\\0.4444 & 0.3333 & 0.2222\end{bmatrix} \tag{5-63}$$

这个矩阵的每行都收敛到同一个行向量，这就是这个马尔可夫链的稳态分布。

$\boldsymbol{A}_{ij}^{\infty}=P_{ij}(\infty)$＝从状态 i 出发经过无数步后处于状态 j 的概率，对于固定的 j，这个值是一样的。

换句话说，它不依赖于开始的状态，因为稳态分布是整个马尔可夫链的属性，它不依赖于开始的状态。只有在满足一定的条件下，A^{∞} 的行才会收敛。需要满足不可约性和周期性条件。

5.5.2　马尔可夫链的性质

本节对马尔可夫链的 4 个性质(不可约性、常返性、周期性和遍历性)进行简单描述。与马尔可夫性质不同，这 4 个性质在状态转移时有所体现，并非马尔可夫链天然拥有。

1. 不可约性

如果一个马尔可夫链的状态空间只有一个连通类，即状态空间的全体成员在一个连通类中，则该马尔可夫链是不可约的，其具有不可约性，否则马尔可夫链具有可约性。马尔可夫链具有不可约性意味着随机变量可在任意状态间转移。

2. 常返性

若马尔可夫链在到达一种状态后，能在之后的演变中返回该状态，则该状态是常返状态，或该马尔可夫链具有常返性，否则马尔可夫链具有瞬变性。对状态空间的某种状态 i，马尔可夫链对某一给定状态的返回时间是其所有返回时间的下确界：

$$T_i=\inf\{n>0:X_n=s_i\mid X_0=s_i\},\quad T_i=\infty \text{ 如果 } \forall n>0:X_n\neq s_i \tag{5-64}$$

若 $T_i=\infty$，则该状态不存在瞬变性或者常返性。

若 $T_i<\infty$，则状态 i 的瞬变性和常返性按以下准则判断：

若

$$\sum_{n=1}^{\infty}p(T_i=n)<1 \tag{5-65}$$

则该状态 i 具有瞬变性。

若

$$\sum_{n=1}^{\infty}p(T_i=n)=1 \tag{5-66}$$

则该状态 i 具有常返性。

此外，常返性的状态可细分为正常返状态和零常返状态。我们通过计算其平均返回时间来判断。

$$E(T_i)=\sum_{n=1}^{\infty}n\cdot p(T_i=n) \tag{5-67}$$

若平均返回时间 $E(T_i)<\infty$，则该状态 i 是正常返的，否则为 0 常返的。

若有限种状态的马尔可夫链是不可约的，则其所有状态必是正常返的。

3. 周期性

一个正常返的马尔可夫链可能具有周期性，即处于某一状态 i 的马尔可夫链可于演变

过程中按某一大于 1 的周期返回该状态。

4. 遍历性

若马尔可夫链的一种状态是正常返的和非周期的，则该状态具有遍历性。

5. 案例

下面将用随机漫步和赌徒破产这两个例子帮助读者理解马尔可夫链。

1）随机漫步

随机漫步有多种定义方式，书中采用一种比较常见的定义。

一种状态为整数 $i=0,\pm1,\pm2\cdots$ 的马尔可夫链，如果对于一个给定的 $0<p<1$，则它的转移概率满足下式：

$$P_{i,i+1}=p=1-P_{i,i-1},\quad i=0,\pm1,\pm2,\cdots \tag{5-68}$$

则称它为随机漫步。

随机漫步的转移图如图 5-5 所示。

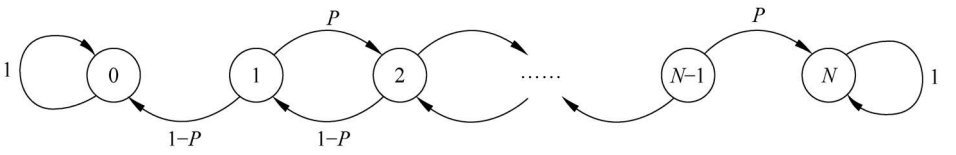

图 5-5　随机漫步的转移图

转置矩阵如下：

$$\begin{bmatrix} \ddots & 0 & 0 & & \\ 1-P & 0 & P & & \\ & 1-P & 0 & P & \\ & 0 & 1-P & 0 & P \\ & & & & \ddots \end{bmatrix} \tag{5-69}$$

2）赌徒破产

赌徒破产是随机漫步的特例。假设有一个赌徒，在每次下注中有 p 的概率赢一块钱，有 $1-p$ 的概率输一块钱。他会一直下注直到输光所有的钱或者资产达到 N。于是可以知道，这是一个马尔可夫链，它有着下面的转移概率。

$$\begin{aligned} &P_{i,i+1}=p=1-P_{i,i-1},\quad i=1,2,\cdots,N-1 \\ &P_{00}=P_{NN}=1 \end{aligned} \tag{5-70}$$

赌徒破产的转移图如图 5-6 所示。

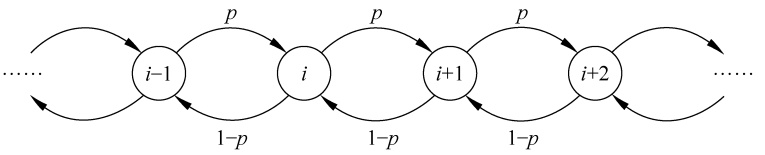

图 5-6　赌徒破产的转移图

转置矩阵如下：

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 1-p & 0 & p & & \\ 0 & 1-p & 0 & p & \\ \vdots & & & \ddots & \\ 0 & & & & 1 \end{bmatrix} \tag{5-71}$$

5.5.3　泊松过程与 Cox 过程

随机过程是处理包含时间及数据序列的概率模型，例如随机过程可用于每天的股票价格数据序列建模。

序列中的每个数据都被视为一个随机变量，所以简单地说，随机过程就是一串（有限或者无限）随机变量序列，与概率的基本概念没有本质的区别。设在某个试验的样本空间中的每个试验结果，对应着一个数列，这个数列中的每个数都对应着一个随机变量。

但是，随机过程和随机变量序列有明显的区别，主要表现在以下几方面：

（1）更倾向于强调过程中产生的数据序列之间的相关关系，例如股票的未来价格与历史价格是什么关系？

（2）对整个过程中的长期均值感兴趣。例如，有多大比例的时间，机器处于闲置？

（3）有时需要刻画某些边界事件的似然或者频率。例如在给定时间内，电话系统里所有的电路同时处于忙碌状态的概率是多少？计算机网络中缓冲器数据溢出的频率是多少？

随机过程的种类非常多，本书只讨论泊松过程。统计学上也称泊松过程为点过程。

泊松过程是连续时间轴上的到达过程。通常，当一个到达过程在应用上无法将连续时间离散化时，就采用泊松过程来刻画。泊松过程是伯努利过程的连续版本。现在考虑连续型的到达过程，即任意的实数 t 都有可能是到达时刻。定义：

$$P(k,\tau)=P(\text{在时间段长度为 } \tau \text{ 的时间内有 } k \text{ 个到达})$$

注意这个定义的内涵，它没有指明区间的位置，这意味着，不管这个位置在哪，只要时间区间的长度为 τ，这个区间内的到达数的分布律就是：

$$P(k,\tau),\quad k \in \mathbf{N} \tag{5-72}$$

此外，还要介绍一个正参数 λ，称为过程的“到达率”或者“强度”。

在一切开始之前，先介绍二项分布的泊松近似。

参数为 λ 的泊松分布的随机变量 Z 取非负整数值，其分布如下：

$$p_Z(k)=\mathrm{e}^{-\lambda}\frac{\lambda^k}{k!},\quad k \in \mathbf{N} \tag{5-73}$$

均值和方差是

$$E[Z]=\lambda,\quad \mathrm{var}[Z]=\lambda \tag{5-74}$$

当 $n\to\infty$，$p=\dfrac{\lambda}{n}$ 时，二项分布的概率：

$$p_S(k)=\binom{n}{k}\cdot p^k(1-p)^{n-k} \tag{5-75}$$

其中，$\binom{n}{k}$为组合数，可以写为$\frac{n!}{(n-k)!k!}$

泊松过程的定义。

具有下述三条性质的到达过程被称为参数为 λ 的泊松过程：

(1)(时间同质性)k 次到达的概率 $P(k,\tau)$在相同长度 τ 的时间段内都是一样的。

(2)(独立性)一个特定时间段内到达的数目与其他时间段内到达的历史是独立的。

(3)(小区间概率)概率(k,τ)满足以下关系：

$$\begin{aligned}P(0,\tau)&=1-\lambda\tau+o(\tau)\\P(1,\tau)&=\lambda\tau+o_1(\tau)\\P(k,\tau)&=o_k(\tau),\quad k=2,3,\cdots\end{aligned} \tag{5-76}$$

这里 τ 的函数$o(\tau)$和 $o_k(\tau)$满足：

$$\lim_{\tau\to 0}\frac{o(\tau)}{\tau}=0,\quad \lim_{\tau\to 0}\frac{o_k(\tau)}{\tau}=0 \tag{5-77}$$

第 1 个性质，称为“到达”在任何时候都是“等可能”的。在任何长度为 τ 的时间段内，到达数具有相同的统计性质，即具有相同的分布律。这与伯努利过程中的假设(对所有的试验，成功的概率都是 p)是相对应的。

为了解释第 2 个性质，考虑一个时间长度为 $t'-t$ 的特殊区间$[t,t']$。在这段时间段里，发生了 k 次到达的无条件概率是 $P(k,t'-t)$。假设我们手里有这个区间之外的完全或者部分到达的信息。那么(独立性)是说，这个信息是无用的：在$[t,t']$内发生了 k 次到达的条件概率仍是无条件概率 $P(k,t'-t)$。这个性质类比于伯努利过程的试验独立性。

第 3 个性质非常关键。$o(\tau)$和 $o_k(\tau)$项是指它们相对 τ 而言，当 τ 非常小的时候，是微不足道的。可以将这些余项理解为 $P(k,\tau)$做泰勒展开时，展开式中的 $O(\tau^2)$项，所以对非常小的 τ，到达一次的概率大致是 $\lambda\tau$，加上一个微不足道的项，类似地，对非常小的 τ，没有到达的概率是 $1-\lambda\tau$，到达两次或更多次的概率与 $P(1,\tau)$相比是可以忽略的。

区间内到达的次数：

现在开始推导泊松过程中与到达相关的概率分布。首先与伯努利过程建立联系来计算一个区间内到达次数的分布列。先考虑一个固定的长度为 τ 的时间区间，将它分成$\frac{\tau}{\delta}$的小区间，每个小区间的长度为 δ，δ 是一个非常小的数，由（小区间概率）性质可知，任意一个小区间内有两次或更多次到达的概率是非常小的，可以忽略不计，而且由（独立性）性质可知，不同的时间段到达的状况又是相互独立的。更进一步地，在每个小区间内，到达一次的概率大致是 $\lambda\delta$，没有到达的概率大致是 $1-\lambda\delta$，所以这个过程可以大致由伯努利过程来近似。当 δ 越来越小时，这个近似就会越来越精确。

在时间 τ 内到达 k 次的概率 $P(k,\tau)$近似地等于以每次实验成功概率为 $p\to\lambda\delta$，进行

$n\to\frac{\tau}{\delta}$次独立伯努利试验，而成功 k 次的(二项)概率。现在保持 τ 不变，令 δ 趋于 0。注意到，这时时间段数目 n 趋于无穷大，而乘积 np 保持不变，等于 $\lambda\tau$，根据二项分布趋于参数为 $\lambda\tau$ 的泊松分布，于是可以得到如下重要结论：

$$P(k,\tau)=\mathrm{e}^{-\lambda\tau}\frac{(\lambda\tau)^k}{k!},\quad k\in\mathbf{N} \tag{5-78}$$

由 $\mathrm{e}^{-\lambda\tau}$ 的泰勒展开可以得到：

$$\begin{aligned}&P(0,\tau)=\mathrm{e}^{-\lambda\tau}=1-\lambda\tau+o(\tau),\\&P(1,\tau)=\lambda\tau\mathrm{e}^{-\lambda}=\lambda\tau-\lambda^2\tau^2+o(\tau^3)=\lambda\tau+o_1(\tau)\end{aligned} \tag{5-79}$$

其中，$N\tau$ 表示在长度为 τ 的时间段中到达的次数。这是因为我们考虑的是参数为 $n=\frac{\tau}{\delta}$和 $p=\lambda\delta$ 的二项分布的极限分布，均值为 $np=\lambda\tau$，方差为 $np(1-p)\approx np=\lambda\tau$。

现在推导首次到达时间 T 的概率规律。假设起始时间为 0，则 $T>t$ 当且仅当在时间 $[0,t]$内没有一次到达，所以：

$$F_T(t)=P(T\leqslant t)=1-P(T>t)=1-P(0,t)=1-\mathrm{e}^{-\lambda t},\quad t\leqslant 0 \tag{5-80}$$

然后对 T 的分布函数求导，得到概率密度函数公式：

$$f_T(t)=\lambda\mathrm{e}^{-\lambda t},\quad t\geqslant 0 \tag{5-81}$$

这就说明首次到达时间服从参数为 λ 的指数分布。

泊松过程相关的随机变量及其性质：

服从参数为 $\lambda\tau$ 的泊松分布，这是泊松过程的强度为 λ，在时间长度为 τ 的区间内到达的总次数 N_τ 的分布，它的分布列、期望和方差分别是：

$$p_{N_\tau}(k)=P(k\cdot\tau)=\mathrm{e}^{-\lambda\tau}\frac{(\lambda\tau)^k}{k!},\quad E[N_\tau]=\lambda\tau,\quad \mathrm{var}[N_\tau]=\lambda\tau \tag{5-82}$$

服从参数为 λ 的指数分布这是首次到达的时间 T 的分布，它的分布列、期望和方差是

$$f_T(t)=\lambda\mathrm{e}^{-\lambda t},\quad t\geqslant 0,\quad E[T]=\frac{1}{\lambda},\quad \mathrm{var}[T]=\frac{1}{\lambda^2} \tag{5-83}$$

【重要结论】 独立泊松随机变量之和仍是泊松随机变量。

对任意给定的时间 $t>0$，时间 t 之后的过程也是泊松过程，而且与时间 t 之前(包括时间 t)的历史过程相互独立。

对任意给定的时间 t，令 $\overline{T}$ 是时间 t 之后首次到达的时间，则随机变量 $\overline{T}-t$ 服从参数为 λ 的指数分布，并且与时间 t 之前(包括时间 t)的历史过程相互独立。

上述时间 t 的历史过程相互独立是因为从时间 t 开始的过程满足泊松过程定义的性质。未来与过去的独立性直接来源于泊松过程定义中的独立性假设。$\overline{T}-t$ 具有相同的指数分布，这是因为

$$P(\overline{T}-t>s)=P(\text{在时间}[t,t+s]\text{没有到达})=P(0,s)=\mathrm{e}^{-\lambda s} \tag{5-84}$$

这就是无记忆性。

【案例 5-1】 假设收电子邮件是一个强度为每小时 $\lambda=0.2$ 封的泊松过程。每小时检查一次电子邮件，那么接到 0 封和 1 封电子邮件的概率是多少？

可以使用泊松分布 $\frac{e^{-\lambda\tau}(\lambda\tau)^k}{k!}$ 来计算，这里 $\tau=1$，$k=0$ 或 $k=1$：

$$P(0,1)=e^{-0.2}=0.819,\quad P(1,1)=0.2e^{-0.2}=0.164 \tag{5-85}$$

又假设一天都没有检查电子邮件。那么一封电子邮件都没有收到的概率是多少？再次用泊松分布来计算，即

$$P(0,24)=e^{-0.2\times24}=0.0083 \tag{5-86}$$

还可以这样想，在一天 24 小时里都没有收到信息，那么连续 24 个 1 小时都没有收到信息，而后者 24 个事件都是相互独立的，而且每个时间发生的概率是 $P(0,1)=e^{-0.2}$，所以：

$$P(0,24)=(P(0,1))^{24}=(e^{-0.2})^{24}=0.0083 \tag{5-87}$$

这个结果与上面的结果一致。

【案例 5-2】 进入银行，你会发现有 3 个营业员正在服务客户，而且没有其他人在排队等待。假设你的服务时间和正在服务的客户的服务时间都是具有相同参数的指数分布，并且相互独立。那么你是最后一个顾客离开银行的概率是多少？

答案是 1/3。从你开始接受一名营业员服务的那一刻算起，另两名正在接受服务的顾客还需要的服务时间，与你所需要的服务时间具有相同的分布。另外两位顾客虽然比你早接受服务，但由于泊松过程的无记忆性，他们与你处于同一起跑线上，不算以前的服务时间，三人所需的服务时间的分布是相同的，所以和其他两人具有相同的概率最后离开银行。

泊松过程的简单总结：

开始于一串相互独立并且公共参数为 λ 的指数随机变量序列 $T_1,T_2,\cdots$，它们是相邻到达时间。过程的到达时间为 $T_1,T_1+T_2,T_1+T_2+T_3$ 等。这样形成的随机过程就是泊松过程。

【Cox 过程定义】

Cox 过程是指双随机泊松过程，是对一般的泊松过程或者其他的技术过程的一种推广。是由 Cox 在 1955 年发表的 *Some Statistical Methods Connected with Series of Events* 一文中提出。

实际上，Cox 过程只是双重随机过程中的一类，2010 年由 Ng 和 Tang 等提出的一类技术过程，是一般更新过程的双重随机过程，感兴趣的读者可查看 *Precise large deviations for sums of random variables with consistently varying tails* 一文。一般的泊松过程认为强度为一定常数，不具有随机性，双随机过程是它允许强度为一随机变量，很多学者把它称为 Cox 过程。那么也就更符合实际情况。

【定义 5-1】

满足以下条件的随机过程 $\{A(t),t\geqslant0\}$ 称为一个随机测度。

(1) $A(0)=0$

(2) $A(t)<\infty,\forall t<\infty$

(3) 对于时间 t 而言，$A(t)$是一个单调不减的右连续函数。

【定义 5-2】

其强度为单位强度，即 $\lambda=1$ 的泊松过程$\{N_1(t),t\geqslant 0\}$称为标准泊松过程。

结合以上两个定义，可以得出 Cox 过程的定义，即

令$\{(t),t\geqslant 0\}$为一随机测度，$\{N_1(t),t\geqslant 0\}$并且 $A(t)$和 $N_1(t)$是相互独立的，则计数过程 $N(t)=N_1(t)A(t)=N_1(A(t)),t\geqslant 0$ 称为 Cox 过程，其中 $A(t)=\int_0^t\lambda(s)\mathrm{d}s$，$\lambda(s)$为强度过程。满足所有的 $s\geqslant 0,\lambda(s)>0$，又称 $A(t)$为累积强度过程。

5.5.4　鞅参考价格

鞅是现代金融理论的核心工具。鞅理论是根据观测到的发展趋势来对时间序列进行分类。如果一个随机过程的路径没有展示出明显的趋势或周期，则它就是鞅。平均而言，呈上升趋势的随机过程被称为下鞅，而上鞅则代表了平均程度上呈递减趋势的随机过程。

假设观察一个以时间 t 为指标的随机变量集族。时间是连续的，面对的则是连续时间的随机过程，将观察到的过程记作$\{S_t,t\in[0,\infty]\}$。用$\{\boldsymbol{I}_t,t\in[0,\infty]\}$代表决策者随时间变化可以连续获得的信息集族。当 $s<t<T$ 时，该信息集族满足 $\boldsymbol{I}_s\subseteq\boldsymbol{I}_t\subseteq\boldsymbol{I}_T$，将$\{\boldsymbol{I}_t,t\in[0,\infty]\}$称为滤子。

在讨论鞅理论时，有时需要考虑随机过程在某些特定时间点上的取值。通常选取一列$\{t_i\}$，使其满足：

$$0=t_0<t_1<t_2<\cdots<t_{k-1}<t_k=T \tag{5-88}$$

来表示连续时间区间$[0,T]$内的随机价格过程 S_t，在特定的时间点 t_i，价格过程的取值为 S_{t_i}，如果对任意 $t\geqslant 0$，S_t 的值都包含在信息族 $\boldsymbol{I}_t$ 中，则称$\{S_t,t\in[0,\infty]\}$适应于$\{\boldsymbol{I}_t,t\in[0,T]\}$，即已知信息族 $\boldsymbol{I}_t$，就可以得出 S_t 的值。

定义连续时间鞅：使用不同的信息集可以对价格过程$\{S_t\}$得出不同的预测，这些预测值可以用条件期望来表示。特别地：

$$E_t[S_t]=E[S_T\mid\boldsymbol{I}_T] \tag{5-89}$$

是在 t 时刻利用已知信息预测得到的 S_t 的未来值 S_T 的正式表示。$E_u[S_t]$，$u<t$，表示用到 u 时刻为止的更小的信息集对相同变量 S_t 做出的预测。

如果随机过程$\{S_t,t\in[0,\infty]\}$对于 $\forall t>0$ 都满足：

当 $\boldsymbol{I}_t$ 已知时，S_t 已知(S_t 关于 $\boldsymbol{I}_t$ 是适应的)；

非条件预测值有限：$E|S_t|<\infty$；

如果 $E_t[S_T]=S_t$，$\forall t<T$ 的概率为 1，则对于无法被观察到的未来值的最优预测是最近的观察值 S_t。

那么称该随机过程为关于信息集 $\boldsymbol{I}_t$ 和概率 P 的鞅。这里所有的期望 $E[\cdot]$ 都是建立在概率 P 上的。

根据该定义，鞅是在当前信息集的条件下完全无法预测未来变化的随机过程，例如，假

设 S_t 是一个鞅，考虑长度为 $u>0$ 的时间区间上 S_t 所发生的变化的预测值：

$$E_t[S_{t+u}-S_t]=E_t[S_{t+u}]-E_t[S_t] \tag{5-90}$$

但 $E_t[S_t]$ 是对于值已知的随机变量的预测(因为根据定义 S_t 是关于 $\boldsymbol{I}_t$ 适应的)，因此，它等于 S_t。如果 S_t 是鞅，$E_t[S_{t+u}]$ 也就等于 $E_t[S_t]$，这样就得到了：

$$E_t[S_{t+u}-S_t]=0 \tag{5-91}$$

即对于 S_t 在任意 $u>0$ 的时间区间内变化的最优预测值为 0。也就是说，鞅在未来运动方向是无法预测的，这就是鞅过程的基本特征。如果随机过程的轨迹明显具有可认知的长期或短期趋势，则该过程就不是鞅。

注意，这里还要强调鞅定义中非常重要的一点，鞅的定义总是伴随着特定的信息集和特定的概率。如果改变信息的内容或改变与随机过程相关的概率测度，则所考虑的随机过程可能不再是鞅。

反之，给定非鞅的随机过程 X_t，可以通过调整相应的概率测度 P 将 X_t 转换为鞅。

根据上述定义，如果随机过程 S_t 在给定信息集的条件下完全无法预测未来值，则它就是鞅。股票价格或债券价格都不是完全无法预测的。贴现债券的价格被认为是随时间递增的，通常股票价格亦是如此，因此，如果 B_t 表示在 T 时刻($t<T$)到期的贴现债券的价格，则有 $B_t<E[B_u]$，$t<u<T$，显然，贴现债券价格的运动不满足鞅的条件。

类似地，一般而言，一支风险股票的价格 S_t 会一直有着正的预期收益，因此也不会是鞅。对于小区间 Δ 而言，有

$$E[S_{t+\Delta}-S_t]\approx\mu \tag{5-92}$$

这里 μ 是正的预期收益率。

期货和期权也有着相似的结论。例如，期权具有“时间价值”，并且随着时间的流逝，假定其他条件不变，欧式期权价格会下降。这种随机过程称为上鞅。

如果资产价格更可能是下鞅或上鞅，则为什么我们还对鞅这么感兴趣呢？

这是因为虽然大多数金融资产的价格不是鞅，但可以将它们转化成鞅。例如，可以找到一个概率测度为 Q 使债券或股票价格按无风险利率贴现后变为鞅。在这种情况下，对于债券以下等式成立：

$$E_t^Q[\mathrm{e}^{-ru}B_{t+u}]=B_u,\quad 0<u<T-t \tag{5-93}$$

对于股票以下等式成立：

$$E_t^Q[\mathrm{e}^{-ru}S_{t+u}]=S_t,\quad 0<u \tag{5-94}$$

这种方法在衍生证券定价中非常有用。

有两种方法可以将下鞅转换为鞅。第 1 种方法比较直观，可以从 $\mathrm{e}^{-rt}S_t$ 或 $\mathrm{e}^{-rt}B_t$ 中减去预期趋势，这会使原有趋势附近的波动完全无法预测，因此，“变形”后的变量是鞅。这种方法等价于通过分解来得到鞅。事实上，Doob-Meyer 分解意味着，在某些一般条件下，任意连续时间随机过程可以被分解成一个鞅和一个递增(或递减)过程。减去后者即可得到鞅。

第 2 种方法更加实用，这种方法是改变概率测度，而不是直接减去下鞅。也就是说，如果有

$$E_t^P\left[\mathrm{e}^{-ru}S_{t+u}\right]>S_t \tag{5-95}$$

则可以尝试找出一个“等价”概率测度 Q，使在 Q 测度下新的数学期望满足：

$$E_t^Q\left[\mathrm{e}^{-ru}S_{t+u}\right]=S_t \tag{5-96}$$

这时 $\mathrm{e}^{-rt}S_t$ 就变成了鞅。

上述两式的转换，新的概率测度称为等价鞅测度。

选择第 2 种方法将任意随机过程转换为鞅，那么将用到 Girsanov 定理。在金融资产定价中，这种方法比 Doob-Meyer 分解有着更加广泛的应用。

当套利机会不存在时，市场均衡意味着可以找到一种人造的概率测度 Q，使所有正确贴现后的资产价格 S_t 变为鞅：

$$E_t^Q\left[\mathrm{e}^{-ru}S_{t+u}\mid \boldsymbol{I}_t\right]=S_t \tag{5-97}$$

因此，鞅在资产定价的实践中起着非常重要的作用。

然而这并不是鞅非常有用的唯一原因。鞅理论内容非常丰富，这里讨论鞅理论中的一些实用技巧。

用 X_t 表示在滤子$\{\boldsymbol{I}_t\}$和概率测度 Q 下具有鞅性质的资产价格，它满足：

$$E_t^Q\left[X_{t+\Delta}\mid \boldsymbol{I}_t\right]=X_t \tag{5-98}$$

其中，$\Delta>0$ 表示小的时间区间。那么 X_t 在连续时间下将会有哪种类型的轨迹呢？

为了回答这一问题，首先定义鞅差分 ΔX_t：

$$\Delta X_t=X_{t+\Delta}-X_t \tag{5-99}$$

由于 X_t 是鞅，所以：

$$E_t^Q\left[\Delta X_t\mid \boldsymbol{I}_t\right]=0 \tag{5-100}$$

正如前面所讲，这个等式意味着无论时间区间 Δ 多么小，鞅的增量都应该是完全无法预测的。

这种不规律的轨迹按两种方式出现。它们可以是连续的，也可以是跳跃的，连续鞅如图 5-7 所示，右连续鞅如图 5-8 所示。

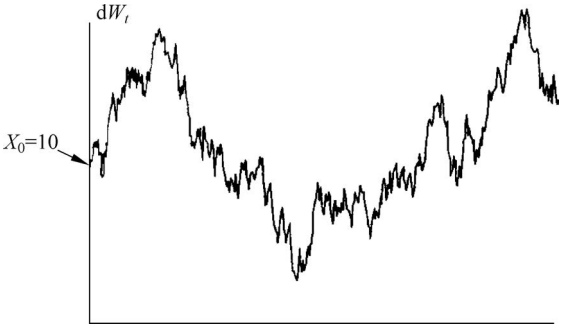

图 5-7　一个连续鞅的例子

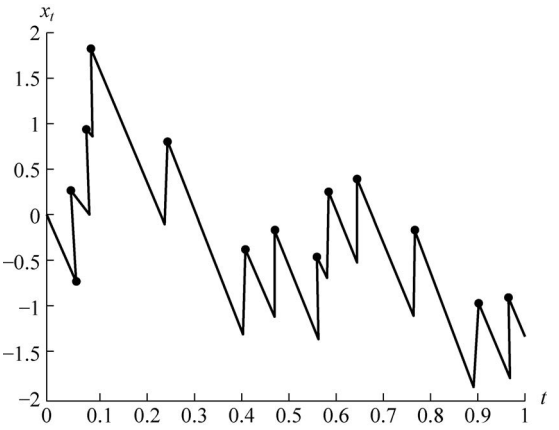

图 5-8　一个右连续鞅的例子

图 5-7 给出了一个连续鞅的例子。注意，它的轨迹是连续的，当 $\Delta \to 0$ 时，有

$$P(\Delta X_t > \varepsilon) \to 0, \quad \forall \varepsilon > 0 \tag{5-101}$$

图 5-8 给出了一个右连续鞅的例子。在这种鞅中，它的路径存在随机的跳跃点。

这种路径的不规律性和跳跃的可能性正是表示资产价格时所需要的理论工具，尤其是在已知套利定理的情况下。

除此之外，鞅还有一些重要的含义。假设 X_t 是连续鞅，并且对任意 $t>0$，X_t 具有有限的二阶矩：

$$E[X_t^2] < \infty \tag{5-102}$$

这种随机过程具有有限的方差，被称为连续平方可积鞅。通过调整时间后的布朗运动来对这种鞅进行表示是很有意义的。也就是说，连续平方可积鞅非常接近于布朗运动。这意味着变化的不可预测性和不存在跳跃是连续时间布朗运动的两个重要性质。

这一点本质上意味着，如果连续平方可积鞅适合于对资产价格进行建模，则可以假设资产价格过程中的微小增量是具有正态性的。

这里使用两个“小区间”Δ 内观察到的相互独立的泊松过程来构建一个鞅。

假设金融市场受“好”消息和“坏”消息的影响。忽略消息的具体内容，仅保留它们是“好”还是“坏”这一信息。N_t^G 和 N_t^B 分别表示到 t 时刻为止出现的“好”消息和“坏”消息的数目。进一步假设消息到达金融市场的方式是与历史数据完全无关的。两种消息之间是相互独立的。

最后，在小区间 Δ 内，至多出现一条消息，并且两种类型的消息出现的概率相同，因此，在区间 Δ 的增量变化 ΔN_t^G 和 ΔN_t^B 的概率分布为

$$P(\Delta N_t^G = 1) = P(\Delta N_t^B = 1) \approx \lambda \Delta \tag{5-103}$$

定义变量 M_t：

$$M_t = N_t^G - N_t^B \tag{5-104}$$

那么 M_t 是鞅。

5.5.5　列维过程

列维过程是所有具有平稳、独立增量的随机过程的统称。列维-辛钦定理根据所包含的过程的特征总结出了列维过程的特征。它告诉我们，存在测度 ν，对所有 $u\in\mathbf{R}$ 和 t 非负，列维过程的特征函数可写为

$$E(\mathrm{e}^{\mathrm{i}uX_t})=\exp(t_{\phi}(u)) \tag{5-105}$$

其中，

$$\phi(u)=\mathrm{i}\gamma u-\frac{1}{2}\sigma^2u^2+\int_{-\infty}^{+\infty}(\mathrm{e}^{\mathrm{i}uy}-1-\mathrm{i}uy1_{\{|y|\leqslant 1\}})\,\mathrm{d}\nu(y) \tag{5-106}$$

这里 γ 和 σ 是实数，ν 是 $\mathbf{R}$ 上的测度，满足 $\nu(0)=0$，并且：

$$\int_{-\infty}^{+\infty}\min(1,x^2)\,\mathrm{d}\nu(x) \tag{5-107}$$

有界。假定列维过程 $\{X_t\}_{t\geqslant 0}$ 为以下形式：

$$X_t=(r-q+\omega)t+Z_t \tag{5-108}$$

该过程包含 ω 控制的漂移项和纯跳跃成分 $\{Z_t\}_{t\geqslant 0}$。在方差伽马过程中，纯跳跃部分的列维测度可以写作 $\mathrm{d}\nu(y)=k(y)\mathrm{d}y$，其中 $k(y)$ 为

$$k(y)=\frac{\mathrm{e}^{-\lambda_p y}}{\nu y}1_{y>0}+\frac{\mathrm{e}^{-\lambda_n|y|}}{\nu y}1_{y<0} \tag{5-109}$$

且

$$\lambda_p=\left(\frac{\theta^2}{\sigma^4}+\frac{2}{\sigma^2}\right)^{\frac{1}{2}}-\frac{\theta}{\sigma^2} \tag{5-110}$$

$$\lambda_n=\left(\frac{\theta^2}{\sigma^4}+\frac{2}{\sigma^2}\right)^{\frac{1}{2}}+\frac{\theta}{\sigma^2} \tag{5-111}$$

5.5.6　GP 模型通俗解读

GP 模型出自 2011 年 Fabien Guilbaud 和 Huyen Pham 的论文 *Optimal High Frequency Trading with limit and market orders*。

AS 模型在实际应用中会有两个缺点，一是这种处理方法涉及最优时间窗口的选择问题，如果处理不谨慎，则可能会造成过度拟合。二是这种方法存在按照最小跳价取整的舍入误差问题，对离散的挂单过程无法准确刻画。

为了解决上述两个问题，GP 模型应运而生。对大跳价资产而言，限价单的队列较长，绝大部分市价单能成交在买一和卖一价上，因此使用市价单的意义并不大，所以 GP 模型最优报价策略的选择只局限在买一和卖一，以及买一加一个跳价或卖一减一个跳价上，后两种挂单方式实际相当于使用市价单了。这里并不存在在更深价位挂单的情景，因此盈利也相当有限，多数时候不会超过半个跳价，可以预期这个策略将对手续费返还要求较高。这个模型的另一效果在于做市过程中产生的库存风险的管理。

GP模型提出的离散模型中包含了3个随机过程：一个用来模拟离散取值的价差S_t，这是一个马尔可夫过程，另外两个是模拟市价买单和市价卖单到达的过程(双随机泊松过程)。

首先，买卖价差S_t可以看作一个只取离散值的随机过程，即

$$S_t:=\{\delta,2\delta,\cdots,m\delta\} \tag{5-112}$$

其中，δ是最小跳价。价差在状态i和状态j之间的转移概率可以定义为

$$P\{S_{t+1}=j\delta \mid S_t=i\delta\}=\rho_{ij},\quad \text{s.t.}\ \rho_{ii}=0 \tag{5-113}$$

这样S_t就是一个时间连续的马尔可夫链，有密度矩阵：

$$\boldsymbol{R}(t)=(r_{ij}(t))_{1\leqslant i,j\leqslant m} \tag{5-114}$$

对于密度矩阵$\boldsymbol{R}(t)$，有

$$r_{ij}=\Lambda(t)\rho_{ij},\quad i\neq j \tag{5-115}$$

$$r_{ii}(t)=-\sum_{j\neq i}r_{ij}(t) \tag{5-116}$$

其中，$\Lambda(t)$为高频数据的采样频率。

在建立价差的随机过程后，接着需要研究的是在不同价差条件下，做市商所挂限价单的成交概率。这里把做市商的报价策略α_t表示为

$$\alpha_t=(Q_t^b,Q_t^a,L_t^b,L_t^a) \tag{5-117}$$

其中，L_t^b,L_t^a分别表示做市商所挂的买单和卖单的数量，Q_t^b,Q_t^a分别表示做市商所挂的买单和卖单的价格。

做市商的买单报价策略池$\pi^b(p,s)$可以用中间价p和价差s表示，即

$$\pi^b(p,s)=\begin{cases}\text{Bb}_+=p-\dfrac{s}{2}+\delta\\ \text{Bb}=p-\dfrac{s}{2}\end{cases} \tag{5-118}$$

这意味着所报买价在买一价加一跳价中选择。

同样，卖单报价策略池$\pi^a(p,s)$可以表示为

$$\pi^a(p,s)=\begin{cases}\text{Ba}_-=p+\dfrac{s}{2}-\delta\\ \text{Ba}=p+\dfrac{s}{2}\end{cases} \tag{5-119}$$

即所报卖价在卖一价减一跳价中选择。

能够与做市商所挂限价单匹配的市价买单和卖单可以用两个相互独立的Cox过程N^a和N^b模拟，它们的密度函数可以表示为$\lambda^a(Q_t^a,S_t)$和$\lambda^b(Q_t^b,S_t)$。这里的λ^a和λ^b分别是在m种价差状态下和两种报价价格上得到的密度，因此λ^a和λ^b在同一时间段上各有$2m$个值。

为了让GP模型能够顺利地运行，需要对模型中的参数给出相应的估计。GP模型使用的参数包括不同价差状态之间的转移概率$P\{S_{t+1}=j\delta \mid S_t=i\delta\}=\rho_{ij}$，以及各种状态在不

同价位挂单的成交密度 $\lambda^a(Q_t^a, S_t)$ 和 $\lambda^b(Q_t^b, S_t)$。

价差从状态 i 至 j 的转移概率 ρ_{ij} 按照下式统计：

$$\hat{\rho}_{ij} = \frac{\sum_{n=1}^{K} 1_{\{\hat{s}_{n+1}, \hat{s}_n = (j\delta, i\delta)\}}}{\sum_{n=1}^{K} 1_{\{\hat{s}_n = i\delta\}}} \tag{5-120}$$

即统计价差转移至不同状态与其所占的时间比例。

不同价差状态下限价单的成交密度 λ 的估计由于要考虑排队效应，所以较为烦琐。我们的估计需要基于一个假设：做市商的挂单都是在买一或者卖一出现变动后才发出的。而且所发出的挂单成交后，如果相应的买一或者卖一价不变，则不再发出新的报价。

以买单举例，当买一价格在 θ_n 时刻出现变动时，做市商能够及时获得相应信息，然后进行报价，所报价格经过一定延时后到达交易所。θ_n 时刻新的买单队列长度为 $V_{\theta_n}^b$，则做市商的挂单所在队列长度为 $V_0 + V_{\theta_n}^b$。在 θ_{n+1} 时刻后买一价格发生改变，从 θ_n 到 θ_{n+1} 时刻发生在该买价上的成交量为 $V_{\theta_{n+1}}^S$，如果 $V_0 + V_{\theta_n}^b < V_{\theta_{n+1}}^S$，则判定做市商所挂买单成交。做市商在买一价挂单的 Cox 过程 N^b 在不同价差状态 i 下可记为

$$\widetilde{N}_{\theta_{n+1}}^{b,Bb,i} = \widetilde{N}_{\theta_n}^{b,Bb,i} + 1\{V_0 + V_{\theta_n}^b < V_{\theta_{n+1}}^s s_{\theta_n} = i\delta\} \tag{5-121}$$

$$\widetilde{N}_0^{b,Bb,i} = 0 \tag{5-122}$$

V_0 用来模拟做市商的交易系统延时，做市商的交易系统越快则 V_0 越小。

如果做市商在买一价上加一最小跳价上挂单，则只需 $V_0 < V_{\theta_{n+1}}^S$ 便可判定挂单成交，相应的 Cox 过程 N^b 可以记为

$$\widetilde{N}_{\theta_{n+1}}^{b,Bb_+,i} = \widetilde{N}_{\theta_n}^{b,Bb,i} + 1\{V_0 < V_{\theta_{n+1}}^s, s_{\theta_n} = i\delta\} \tag{5-123}$$

$$\widetilde{N}_0^{b,Bb_+,i} = 0 \tag{5-124}$$

再把买单队列在不同价差状态 i 下存在的时间记为

$$\widetilde{T}_{\theta_{n+1}}^i = \widetilde{T}_{\theta_n}^i + (\theta_{n+1} - \theta_n) 1\{s_{\theta_n} = i\delta\} \tag{5-125}$$

$$\widetilde{T}_0^i = 0 \tag{5-126}$$

那么不同价差状态 i 下，做市商所报的买单价格成交密度可以表示为

$$\tilde{\lambda}_i^b(q^b) = \frac{\widetilde{N}_{\theta_n}^{b,q^b,i}}{\widetilde{T}_{\theta_n}^i}, \quad q^b \in \mathrm{Bb}, \mathrm{Bb}_+ \tag{5-127}$$

卖单 $\tilde{\lambda}_i^a(q^a)$ 也可以用相同的方式进行统计，但在实际应用中会把两者取平均后使用，这样就不会有买卖偏好了。

利用 GP 模型做市的目的是最大化做市收益及在做市结束时拥有最小的库存，假设做市结束时间 T 时，做市商拥有现金流为 X_T，累积的库存为 Y_T，中间价为 P_T，以及处于差

价状态 S_T，即 $i\delta, i\in 1,2,\cdots,m\}$，做市商的目标是最大化目标函数：

$$v_i(t,x,y,p)=\max E\left[X_T+Y_TP_T-|Y_T|\frac{i\delta}{2}-\gamma\int_0^T g(Y_t)\,\mathrm{d}t\right],\quad i\in s \tag{5-128}$$

其中，$g(y)=y^2$。

使用分离变量法，v_i 可以进一步分解为

$$v_i(t,x,y,p)=x+yp+\phi_i(t,y) \tag{5-129}$$

这个分解可以直观地理解为未来最优策略的选取与已经取得的收益 x 无关，p 是一个鞅(martingale)，期望是 0，所以可以把 x 和 yp 从 (t,x,y,p) 中分离出去。

利用动态规划原理 $\phi_i(t,y)$ 可以通过以下 HJB 方程求得

$$\begin{aligned}&\frac{\partial\phi_i}{\phi t}+\sum_{j=1}^{m}r_{ij}(t)\left[\phi_j(t,y)-\phi_i(t,y)\right]+\\&\sup_{(q^b,l^b)\in Q_i^b\times[0,\overline{l}]}\lambda_i^b(q^b)\left[\phi_j(t,y+l^b)-\phi_i(t,y)+\left(\frac{i\delta}{2}-1_{q^b=\mathrm{Bb}_+}\right)\right]+\\&\sup_{(q^a,l^a)\in Q_i^a\times[0,\overline{l}]}\lambda_i^a(q^a)\left[\phi_j(t,y+l^a)-\phi_i(t,y)+\left(\frac{i\delta}{2}-1_{q^a=\mathrm{Ba}_-}\right)\right]-\gamma g(y)=0\end{aligned} \tag{5-130}$$

同时满足终止条件：

$$\phi_i(T,p)=-|y|\frac{i\delta}{2} \tag{5-131}$$

其中，q^b 和 q^a 为需要计算的买卖报价，l^b 和 l^a 为所挂的买卖单数量，$\overline{l}$ 为最大挂单量，γ 为对库存的惩罚系数。方程可以通过有限差分的欧拉公式进行求解。

5.5.7 基于动态规划方法的高频做市策略模型 GP 模型

在开发之前，先来明确一下这个模型的重要思路。它是一个基于动态规划方法的高频做市策略模型 GP 模型，用于在订单驱动市场中，面对离散报价情况下的做市策略优化。

首先，文章建立了一个做市商在限价盘市场做市的框架。做市商可以选择不同的报价方式，包括最优报价、最优报价加一个最小变动单位报价等。与此对应的是不同报价方式下成交概率也不同。成交执行是随机的，用 Cox 过程模拟。做市商的目标是在有限的时间内通过控制报价和数量，最大化收益并控制风险。

然后使用动态规划原理，建立了一个 HJB 方程组，描述了不同报价方式下的做市收益。通过求解 HJB 方程，可以获得给定仓位下的最优报价策略。这里的关键是报价的离散性，以及不同报价对应的成交强度函数。

求解 HJB 方程后，做市商可以根据当前仓位状态，查询价值函数，产生最优的报价和数量。这构成了一个反馈控制循环。

与传统的 Avellaneda-Stoikov 模型相比，GP 模型更适合模拟和优化订单驱动市场下的

高频做市策略，其考虑了报价的离散性，以及订单在买盘卖盘中的优先级等市场微观结构特征。这使策略更贴近实际市场交易的需求。

使用市场高频数据来估计模型中涉及的一些参数，包括报价变化概率，不同报价下的成交强度，这为模型的落地应用提供了可能。

主要的创新点在于建模报价的离散性，引入了订单优先级概念，并给出了直观的参数估计方法。

建模首先考虑的入手要点如下：

(1) 假设股票中间价跟随一个扩散过程，买卖盘间隙(Spread)是一个马尔可夫连续时间链。

股票中间价 P_t 满足扩散过程 $P_t=\mu(P_t,t)\mathrm{d}t+\sigma(P_t,t)\mathrm{d}W_t$，计算代码如下：

```
#//第 5 章//GP.ipynb
def simulate_P(T, P0, mu, sigma):
    P = [P0]
    for t in range(T):
        dP = mu * P[-1] * dt + sigma * P[-1] * dW(t)
        P.append(P[-1] + dP)
    return P
```

买卖盘间隙 S_t 满足马尔可夫链 $S_t=S_{\hat{N}_t}$，$\hat{N}_t\sim\mathrm{Poisson}(\lambda(t))$，计算代码如下：

```
#//第 5 章//GP.ipynb
def simulate_S(T, S0, rate, Q):
   S = [S0]
   N = poisson_process(rate)
   for t in range(T):
       S.append(Q[S[-1], N[t]])
   return S
```

(2) 交易者可以选择限价单或市价单进行交易。限价单有执行风险，市价单有较高的成本。限价单执行数 N_t^a，N_t^b 满足 Cox 过程，强度为 $\lambda^a(S_t,a_t)$，$\lambda^b(S_t,b_t)$ 的市价单，成本函数为 $c(e_t,P_t,S_t)=e_tP_t+|e_t|S_t/2+\varepsilon$，计算代码如下：

```
#//第 5 章//GP.ipynb
def simulate_trades(S, a, b, Na=0, Nb=0):
   for t in range(T):
       dNa = binomial(lambda_a(S[t], a[t]))        #Cox 过程
       dNb = binomial(lambda_b(S[t], b[t]))
       Na += dNa
       Nb += dNb
   return Na, Nb

def market_order_cost(e, P, S):
    return e * P + abs(e) * S/2 + eps
```

(3) 目标是在有限期内最大化资产终值的期望效用，同时控制仓位。形式化为一个混合正则/脉冲控制问题。目标函数：$J=E\left[U(X_T)-\gamma\int_0^t g(Y_t)\,\mathrm{d}t\right]$；动态规划方程 $\min[-v_t-$

$\mathcal{L}v-\gamma g, v-\mathcal{M}v]=0$,计算代码如下:

```
#//第 5 章//GP.ipynb
def value_function(X, Y, P, S, t):
    数值求解 HJB 偏微分方程
    return v(X, Y, P, S, t)

def optimize(T, v):
    #通过最大化 Hamilton 量来获得最优控制
    return a_star, b_star, e_star
```

(4) 通过动态规划原理推导出 HJB 方程,并在具体效用函数情况下化简。HJB 方程的隐式形式版本在 mean-variance 和 exponential utility 情况下化简为仅依赖仓位和间隙的形式,计算代码如下:

```
#//第 5 章//GP.ipynb
def hjb_pde(v, bv, sv):
    #离散并求解 HJB PDE
    return v

def simplified_hjb(y, s):
    #解决特定实用程序的简化 IDE
    return v(y, s)
```

(5) 给出参数估计方法,并在真实数据上进行策略回测,结果显示优化策略确实改善了信息比率。参数估计 $\hat{\rho}_{ij}=\dfrac{\sum_{n=1}^{N}\mathbb{I}(S_{n+1}=j, S_n=i)}{\sum_{n=1}^{N}\mathbb{I}(S_n=i)}$,$\hat{\lambda}=\dfrac{N_T}{T}$,回测比较信息比率 IR=$\dfrac{E[X_T]}{\sqrt{\text{var}[X_T]}}$,计算代码如下:

```
#//第 5 章//GP.ipynb
def estimate_params(ticks):
    #估计函数
    return rho_hat, lambda_hat

def backtest(strategies):
    #进行回测
    return results

def evaluate_ir(results):
    #计算并比较信息比率
    return IR
```

对于以上这些部分的计算内容应该是首要的建模起步。现在我们来制定一个框架,代码如下:

```
#//第 5 章//main.py:
- 导入需要的包
```

```
- 定义全局配置参数
- 调用数据加载、模型训练、回测、结果评估的流程

data/ticks.csv:
- 存放原始的 tick 数据

lob.py:
- 从 Tick 数据中重建限价买卖盘
- 获取不同挡位的价格和数量数据
- 计算委托队列长度和深度
- 绘制买卖盘时间序列走势

strategy.py:
- 定义交易策略类,包含止损、趋势跟踪、统计套利等策略
- 策略类包含生成订单的逻辑

backtest.py:
- 根据策略逻辑在重建的买卖盘上进行回测
- 记录订单执行的价格、时间和数量
- 计算仓位、收益等时间序列

analysis.py
- 对回测结果进行统计分析
- 绘制订单数量和频次分布
- 绘制仓位和收益时间序列图
- 计算收益评价指标,如夏普比率、胜率等

model/hjb.py:
- 定义 HJB 价值函数和控制域
- 使用有限差分方法求解 HJB 方程
- 输入:模型参数、收益函数、交易成本函数
- 输出:最优控制策略

model/train.py
- 使用历史数据训练模型,确定最优参数
- 将优化后的参数传给 HJB 求解

output/:
- 存储回测和分析结果的图片
main.py:

```python
#导入需要的包
import pandas as pd
from lob import LOB
from strategy import Strategy
from backtest import Backtest
from analysis import Analysis

#定义全局配置参数
TICK_DATA = 'data/ticks.csv'
START_DATE = '20230201'
END_DATE = '20230331'
```
```

```
PRODUCT = ''          #读者可以使用自己想要测试的市场

def main():

    #1. 数据加载
    ticks = load_tick_data(TICK_DATA)

    #2. 模型训练(可选)
    #调用 model/train.py

    #3. 策略回测
    lob = LOB(ticks)
    strategy = Strategy()
    backtest = Backtest(lob, strategy)
    backtest.run(START_DATE, END_DATE)

    #4. 回测分析
    analysis = Analysis(backtest)
    analysis.plot_results()
    analysis.evaluate_performance()

if __name__ == '__main__':
    main()
```

在这个框架下,主要流程包括从 tick 数据中加载并重建限价买卖盘;训练模型确定最优参数(可选);定义交易策略,在重建的买卖盘上进行回测;分析回测结果,评估策略表现。

关键模块包括限价买卖盘重建(LOB)、策略定义(Strategy)、回测引擎(Backtest)、回测分析(Analysis),模块代码如下:

```
#//第 5 章//lob.py

import pandas as pd

class LOB(object):

    def __init__(self, ticks):
        self.ticks = ticks
        self.init_from_ticks()

    def init_from_ticks(self):
        #用 tick 数据重建买卖盘
        self.ask_prices = …
        self.ask_volumes = …
        self.bid_prices = …
        self.bid_volumes = …

    def get_ask_price(self, level):
        #返回某一档的卖价
        return self.ask_prices[level]

    def get_bid_price(self, level):
        #返回某一档的买价
        return self.bid_prices[level]
```

策略部分构建了 HJB 方程，用 solve_HJB 函数求解，得到最优控制策略 phi，然后根据当前仓位 inventory 映射到 phi 上的最优控制，产生报价和数量，在策略模块中实现求解最优报价的关键算法，算法代码如下：

```
#//第 5 章//strategy.py
import numpy as np
from scipy.sparse import spdiags
from scipy.sparse.linalg import spsolve

class HFStrategy:

    def __init__(self, params):
        self.params = params
        #初始化策略参数
        #params:
                #tick_size: 最小变动价位
                #lambda_bid_1: 当前最优买价报价成交概率
                #lambda_bid_2: 当前最优买价上浮一档报价成交概率
                #lambda_ask_1: 当前最优卖价报价成交概率
                #lambda_ask_2: 当前最优卖价下浮一档报价成交概率
                #trans_prob: 价差变动概率矩阵
                #y_max: 最大仓位限制
                #l_max: 单笔最大报单量

    def generate_orders(self, state):

        spread = state['spread']
        mid_price = state['mid_price']
        inventory = state['inventory']

        #根据当前状态计算最优报价和数量
        q_bid, q_ask, l_bid, l_ask = self.optimize(spread, inventory)

        #生成订单
        order = {
            'bid_price': self.get_bid_price(q_bid, mid_price, spread),
            'ask_price': self.get_ask_price(q_ask, mid_price, spread),
            'bid_qty': l_bid,
            'ask_qty': l_ask
        }

        return order


    def optimize(self, spread, inventory):

        #构建 HJB 方程
        def HJB(phi, y):
            #phi 为价值函数，y 为仓位
            #r_ij 为价差转移概率矩阵
            #lambda_i 为不同状态下的成交强度
            #gamma 为仓位惩罚系数
```

```
            d_phi = - r_ij *(phi_j - phi_i)
                    - lambda_i *(phi(y + l_bid) - phi(y))
                    - lambda_i *(phi(y - l_ask) - phi(y))
                    - gamma *g(y)

            return d_phi

        #求解 HJB 方程得到最优控制
        y_grid = np.linspace(-Y_MAX, Y_MAX, NY)
        phi = solve_HJB(HJB, y_grid)

        #根据当前仓位 y 解出最优报价和数量
        y_idx = find_index(y_grid, inventory)
        q_bid, l_bid = get_optimal(phi, y_idx, 'bid')
        q_ask, l_ask = get_optimal(phi, y_idx, 'ask')

        return q_bid, q_ask, l_bid, l_ask


    def is_entry_signal(self, lob):
        #判断是否触发入场信号
        …

    def long_entry_order(self):
        #多头开仓订单 specifics
        …

    def get_bid_price(self, q_bid, mid_price, spread):
        if q_bid == 'Bb+':
            return mid_price - spread/2 + self.tick_size
        else:
            return mid_price - spread/2

    def get_ask_price(self, q_ask, mid_price, spread):
        if q_ask == 'Ba-':
            return mid_price + spread/2 - self.tick_size
         else:
            return mid_price + spread/2



    def solve_HJB(HJB, y_grid):

        #构建 HJB 方程的系数矩阵
        n = len(y_grid)
        h = y_grid[1] - y_grid[0]

        data = [1/h **2 *np.ones(n),
                -2/h **2 *np.ones(n),
                1/h **2 *np.ones(n)]

        diags = [-1, 0, 1]
        A = spdiags(data, diags, n, n)
```

```
        #处理边界条件
        A[0,0] = 1
        A[n-1, n-1] = 1

        #求解线性方程组
        rhs = -HJB(y_grid)
        phi = spsolve(A, rhs)

        return phi

    def get_optimal(phi, y_idx, side):

        #根据 phi 求最优控制
        if side == 'bid':
            q1 = 'Bb'
            q2 = 'Bb+'
            lambda_1 = LAMBDA_BID_1
            lambda_2 = LAMBDA_BID_2
        else:
            q1 = 'Ba'
            q2 = 'Ba-'
            lambda_1 = LAMBDA_ASK_1
            lambda_2 = LAMBDA_ASK_2

        #计算最优报价
        v1 = lambda_1 * (phi[y_idx+1] - phi[y_idx])
        v2 = lambda_2 * (phi[y_idx+1] - phi[y_idx] - tick_size)
        if v1 >= v2:
            q = q1
        else:
            q = q2

        #计算最优数量
        l = L_MAX if phi[y_idx+1] - phi[y_idx] >= 0 else 0

        return q, l
```

这里实现了 HJB 方程的有限差分求解函数 solve_HJB，以及根据价值函数求最优控制的 get_optimal 函数。在 solve_HJB 中，使用了 scipy 库构建了三对角矩阵，并调用线性求解器求解了 HJB 方程。在 get_optimal 中，计算了最优报价和数量，其中引入了一些全局变量表示成交强度和交易设置。回测部分，代码如下：

```
#//第 5 章//backtest.py
import pandas as pd

class Backtest():

    def __init__(self, strategy, lob):
        self.strategy = strategy
        self.lob = lob
        self.trades = []
```

```
    def run(self, start, end):

        cur_time = start
        inventory = 0
        cash = 0

        while cur_time < end:

            #获取当前限价盘状态
            lob_state = self.lob.get_state(cur_time)

            #策略生成订单
            order = self.strategy.generate_order(lob_state)

            if order:
                #提交订单并获取执行信息
                trade = self.submit_order(order, cur_time)

                #更新仓位
                inventory += trade['fill_qty']
                cash -= trade['fill_price'] *trade['fill_qty']

            #移动到下一个时间
            cur_time += self.timestep

    def submit_order(self, order, time):

        lob_price = self.lob.get_lob_price(order['side'], order['price'])
        fill_price = lob_price
        fill_qty = order['quantity']

        trade = {
            'time': time,
            'side': order['side'],
            'price': order['price'],
            'fill_price': fill_price,
            'fill_qty': fill_qty
        }

        self.trades.append(trade)

        return trade
```

这里实现了主要的回测逻辑,包括加载策略和限价盘、循环回测并生成订单及更新状态、提交订单并模拟成交,还有记录每次交易。

统计数据分析部分,分析代码如下:

```
#//第 5 章//analysis.py

import matplotlib.pyplot as plt

class Analysis():
```

```
    #由于读者关心的问题集中在策略部分,可视化问题可以根据自己的偏好去实现,这里给出一
#个思路
    def __init__(self, backtest):
        self.backtest = backtest

    def plot_results(self):
        #画出回测结果图表
        plt.plot(self.backtest.pnl)
        plt.plot(self.backtest.position)
        …

    def evaluate_performance(self):
        #计算评价指标
        sharpe = …
        hit_rate = …
```

对此基于动态规划方法的 GP 模型就介绍到这里,其中策略模块的参数模拟是最为重要的地方。

5.6　订单簿的泊松过程建模

在交易的执行过程中,尤其是对于一些流动性不佳的产品,限价单经常面临着不被成交的风险。这可能使交易员的套利交易或其他交易执行失败。

以交易中对于短期内中间价格预判的需求为导向,根据五档订单簿及其历史行情,构建了一个预测短期中间价格走势的模型,能够对交易员在执行套利或者一般情况下建仓平仓的下单指令(市价单或限价单)及下单时机决策起到辅助作用。该方法从相对价格挡位上的订单事件发生遵循泊松过程及短时间泊松率不发生明显变化这两个假设出发,通过严密的逻辑推理,使用 Laplace 变换,连分数和复数域上的数值逆变换,得到上述概率。模型具有四大特点:适用范围广;逻辑严密;响应速度快;可解释性好。可运用于各种订单驱动型市场的高频预测,让交易员在执行交易时能获得更优成交价格,降低订单成交的不确定性。

5.6.1　文献综述

预测短期内中间价格变动的方法之一是使用高频信息,订单簿是一个我们可得的高频信息。

通过订单簿可以部分还原出信息簿。订单簿的动态建模主要有两种方法,一种是经典计量经济学方法,另一种是机器学习方法。计量经济学方法是一种经典的主流研究方法,例如研究价差分析的 MRR 分解、Huang 和 Stoll 分解等,研究订单持续期的 ACD 模型,研究价格预测的 Logistic 模型,如 2013 年的 *Price Jump Prediction in Limit Order Book*。机器学习在金融领域的学术研究也非常活跃,例如 2012 年的 *Forecasting trends of high-frequency KOSPI200 index data using learning classifiers* 是一种常见研究思路,利用技术分析常见的指标(MA、EMA、RSI 等),引入机器学习的分类方法进行市场预测,但这种做

法对订单簿动态信息挖掘不足,也就是说,利用订单簿动态信息进行高频交易的研究还比较少,这是很值得深入研究的领域。在应用层面,海外已经有一些量化模型框架被投资者和量化交易员用来优化他们的交易执行策略(Alfonsi 等,2010, Obizhaeva 和 Wang 2006)。

使用随机过程模型对于限价订单簿建模,用概率去衡量不确定性,以解决交易执行优化问题。

5.6.2 问题引入

在订单的执行过程中,交易员所下的限价单不一定能立即被成交,因此,研究中间价格在短时间的变动,以及限价单成交的可能性十分必要。当模型很确信中间价格方向变动对于执行交易有利时,例如,要在一个产品上建立一个多头头寸而模型预测中间价格大概率会下跌时,可以大胆地用限价单在最佳报价上挂单等待成交,因为那意味着不成交风险小却可以获得更优的价格。

当模型很确信中间价格方向变动对于执行交易不利时,例如,要在一个产品上建立一个多头头寸而模型预测中间价格大概率会上涨时,交易员应立即下一个市价单,避免在中间价格上涨时,要么是对面最佳卖价上涨使建仓成本提高,要么是不得不接受盘口买一价的提高,而买一价的提高会导致两个问题,第一,如果在当前的买一价下限价单,则由于价格优先,后来的上涨后的买一价上的订单会使限价单更难成交;第二,如果为了更可能成交而选择跟随买一价的提高,则同样要付出更高的建仓成本。

5.6.3 限价订单簿随机过程模型

采用的模型由 Rama Cont 等提出。限价订单簿的五档盘口历史数据部分揭示了市场的微观结构,以及买卖双方互动通过复杂的动态博弈导致价格运动的过程。在一个订单驱动型(Order Driven)的市场,从动力学的角度讲,行情的所有演化过程都能由订单簿(Order Book)自下而上、精确完备地决定。逐笔成交数据的信息含量非常丰富。对限价订单簿进行随机过程建模,对订单的产生和成交的预测提供一个以演绎推理为基础的逻辑清晰透明的方法,对于保证样本外的预测精度,模型的可解释性,以及模型对不同市场环境的适应性均有意义。

对高频订单簿的动态过程的建模,可以通过当前订单簿状态去预测它的短期行为,因此需要特别关注,基于当前订单簿状态的各种事件的条件概率。

限价订单簿的动态过程在很多方面类似于一个排队系统。限价单在等待序列中等待被市场单执行或被取消。类比地来看,将限价订单簿建模为一个连续马尔可夫过程,其状态描述了限价单在每个价格挡位的数量。这个模型平衡了 3 个交易员想要达到的特性,一是它容易通过高频数据计算得到;二是它基于订单簿的实证特征;三是它由演绎分析得到,具有良好的可解释性。尤其是这个模型的简洁性,使通过前述的排队系统的相关技巧进行 Laplace 变换及逆变换计算订单簿上各种事件的条件概率成为可能。这些条件概率包括 3 点:①基于当前订单簿状态下,中间价格下一次变动是上涨还是下跌的概率;②在中间价

格变动之前我们的订单被成交的概率；③当买一价和卖一价只相距一个最小价格挡位时，限价单在对面报价反向移动之前被成交的概率。

模型的参数可以很容易地从订单簿的历史数据中获得，各种概率也可以高效地被计算出来，以捕捉稍纵即逝的交易机会。

在这个可以程式化地描述订单驱动型市场中限价订单簿动态过程的模型中，订单流被描述为一系列的相互独立的泊松过程，其参数估计方法后文将进行介绍。

用连续时间马尔可夫过程对订单簿进行建模，$X(t)\equiv(X_1(t),\cdots,X_n(t))_{t\geqslant 0}$，这里$|X_p(t)|$是在价格 p 处的订单，$1\leqslant p\leqslant n$，其中$-X_p(t)$用来描述买单数量，$X_p(t)$ 用来描述卖单数量。$\{1,\cdots,n\}$为价格挡位，在大部分市场中，价格是离散的，将可能的价格挡位映射到$\{1,\cdots,n\}$ 上。那么 t 时刻的卖价就是 $p_A(t)\equiv\inf\{p=1,\cdots,n,X_p(t)>0\}\wedge(n+1)$，买价就是 $p_B(t)\equiv\sup\{p=1,\cdots,n,X_p(t)<0\}\vee 0$，定义中的取小和取大是为了应对没有订单的边界情况。

中间价格 $p_M(t)$被定义为 $p_M(t)\equiv\dfrac{p_B(t)+p_A(t)}{2}$，而点差或者称为买卖价差 $p_S(t)$被定义为 $p_S(t)\equiv p_A(t)-p_B(t)$。

由于大部分的交易活动发生在最优买价和最优卖价的附近，因此，我们定义相对价格下的订单数。这个设定将同时有助于交易员通过相对价格下订单的产生与消失的过程来模拟一个变动中的订单簿而不限定于特定的价格范围。

$$Q_i^B(t)=\begin{cases}X_{P_A(t)-i}(t) & 0<i<p_A(t)\\ 0 & p_A(t)\leqslant i<n\end{cases}\tag{5-132}$$

$$Q_i^A(t)=\begin{cases}X_{p_B(t)+i}(t) & 0<i<n-p_B(t)\\ 0 & n-p_B(t)\leqslant i<n\end{cases}\tag{5-133}$$

分别将其定义为距离最优卖价为 i 的买价挡位上的订单数，与距离最优买价为 i 的卖价挡位上的订单数。相对价格挡位的计算和它上面的订单数如图 5-9 所示。

	价	量				相对价	相对价格挡位
卖五	2.9325	2		2.9325	-2.8975	0.035	14 档位
卖四	2.91	5		2.91	-2.8975	0.0125	5 档位
卖三	2.9075	2		2.9075	-2.8975	0.01	4 档位
卖二	2.905	1		2.905	-2.8975	0.0075	3 档位
卖一	2.9025	5		2.9025	-2.8975	0.005	2 档位
买一	2.8975	11		2.8975	-2.9025	-0.005	-2 档位
买二	2.895	3		2.895	-2.9025	-0.0075	-3 档位
买三	2.89	2		2.89	-2.9025	-0.0125	-5 档位
买四	2.885	2		2.885	-2.9025	-0.0175	-7 档位
买五	2.8825	3		2.8825	-2.9025	-0.02	-8 档位

图 5-9　FR007.5Y.IRS 的五档订单簿(2019/6/20 14:00:05)

当有新的订单到来，或者已有的订单被撤销或者成交时，$x^{p\pm 1}\equiv x\pm(0,\cdots,1,\cdots,0)$，其中代表订单簿状态的 $x\in\mathbf{Z}^n$ 并且 $1\leqslant p\leqslant n$，并且其中的 1 在第 p 位置。那么有

一个在价格 p 其中 $p<p_A(t)$ 的限价买单将会使在价格 p 处的订单增加，订单簿状态

$x \to x^{p-1}$；

一个在价格 p 其中 $p > p_B(t)$ 的限价卖单将会使在价格 p 处的订单增加，订单簿状态 $x \to x^{p+1}$；

一个在市场价买单将会使在价格 $p_A(t)$处的订单减少，订单簿状态 $x \to x^{p_A(t)-1}$；

一个在市场价卖单将会使在价格 $p_B(t)$处的订单减少，订单簿状态 $x \to x^{p_B(t)+1}$；

一个在价格 p 其中 $p < p_A(t)$ 的限价买单撤销将会使在价格 p 处的订单减少，订单簿状态 $x \to x^{p+1}$；

一个在价格 p 其中 $p > p_B(t)$ 的限价卖单撤销将会使在价格 p 处的订单减少，订单簿状态 $x \to x^{p-1}$。

订单簿状态的变化由市价单的到来及流入各个价格挡位的限价单和订单取消驱动，这些订单流可以被计数过程描述。到来的订单流的速率的重要决定因素是订单流对应的价格挡位距离最佳买价或者卖价的距离的远近，在最佳买价或者卖价附近订单流到来的通常会快一些。

为了描述这些实证特点同时不失去模型的解析性(这对于高频数据的应用及所需的各种条件概率的计算十分重要)，使用一个随机过程模型来描述之前所述的市场价单、限价单和订单取消事件，假设它们遵循独立泊松过程。更精确地说，假设对于 $i \geqslant 1$，距离最佳卖价为 i 的限价买单的到来的时间间隔为相互独立的指数分布，速率为 $\lambda_B(i)$；距离最佳买价为 i 的限价卖单的到来的时间间隔为相互独立的指数分布，速率为 $\lambda_A(i)$；市场价卖单的到来的时间间隔为相互独立的指数分布，速率为 μ_A；市场价买单的到来的时间间隔为相互独立的指数分布，速率为 μ_B；距离最佳卖价为 i 的限价买单的撤单速率与当前该价格挡位上的限价买单的数量成比例：如果当前价格挡位上的订单数量为 x，则撤单的速率为 $\theta_B(i)x$。这个假设可以这样理解，如果有 x 个当前挡位的订单，每个订单的取消都服从参数为 $\theta_B(i)$ 的指数分布，则这一挡位的订单总的撤销速率为 $\theta_B(i)x$。

距离最佳买价为 i 的限价卖单的撤单速率与当前该价格挡位上的限价卖单的数量成比例：如果当前价格挡位上的订单数量为 x，则撤单的速率为 $\theta_A(i)x$。这个假设可以这样理解，如果我们有 x 个当前挡位的订单，每个订单的取消都服从参数为 $\theta_A(i)$的指数分布，则这一挡位的订单总的撤销速率为 $\theta_A(i)x$。

把这些事件的到来的速率建模为一个关于距离当前最佳买价/卖价距离 i 及时间 t 的函数 λ_t：$\{1,\cdots,n\} \to [0,\infty)$。

基于以上假设，X 是一个连续马尔可夫过程，状态空间为 $\mathbf{Z}^n$，转移概率定义为

$x \to x^{p-1}$ 以速率 $\lambda_{B,t}(p_A(t)-p)$ 发生，其中 $p < p_A(t)$；

$x \to x^{p+1}$ 以速率 $\lambda_{A,t}(p-p_B(t))$ 发生，其中 $p > p_B(t)$；

$x \to x^{p_B(t)+1}$ 以速率 μ_B 发生；

$x \to x^{p_A(t)+1}$ 以速率 μ_A 发生；

$x \to x^{p+1}$ 以速率 $\theta_B(p_A(t)-p)|x_p|$ 发生，其中 $p<p_A(t)$；

$x \to x^{p-1}$ 以速率 $\theta_A(p-p_B(t))|x_p|$ 发生，其中 $p>p_B(t)$。

现在有了对于限价订单簿的模型，并且提到了上述的多种泊松过程的速率，数据包括打上了时间戳的成交信息（包括成交价格和成交量）和五挡报价信息（包括价格和量），对于这些速率参数的估计方法如下：

限价单的到来速率函数可以通过 $\hat{\lambda}_t(i)=\dfrac{N_l(i)}{T_\times}$ 来估计，其中 $N_l(i)$ 是样本中距离当前最佳买价/卖价距离 i 的限价单到达的总数，$T_\times$ 是总的交易时间。$N_l(i)$ 是通过统计样本中距离当前最佳买价/卖价距离 i 的限价单报价增加的量得到。

市场单的到来速率函数可以通过 $\hat{\mu}_t(i)=\dfrac{N_m(i)}{T_\times}$ 来估计，其中 $N_m(i)$ 是样本中距离当前最佳买价/卖价距离 i 的市场单的总数，$T_\times$ 是总的交易时间。$N_m(i)$ 是通过统计样本中距离当前最佳买价/卖价距离 i 的成交的量得到。

为了估计订单的取消速率，首先估计距离最优卖价/买价为 i 的价格挡位上的订单数的稳定数量 Q_i，计算其在订单簿 tick 频率下的平均量，若共有 M 个订单更新状态，则 $Q_i=\dfrac{\sum_{j=1}^{M}Q_{i,j}}{M}$。那么撤销率函数可以估计为 $\hat{\theta}_t(i)=\dfrac{N_c(i)}{T_\times Q_i}$，其中 $N_c(i)$ 是样本中距离当前最佳买价/卖价距离 i 的限价单的撤销的总数，$T_\times$ 是总的交易时间。$N_c(i)$ 是通过统计样本中距离当前最佳买价/卖价距离 i 的限价单报价减少的量减去市价单的量得到。

速率均分别对买单和卖单进行计算，以获得最符合实证检验的参数。同时，计算的采样窗口选择与实际产品的市场活跃度及规律延续的时间有关。市场越活跃，所需的周期便越短（否则易出现总计数量为 0 导致速率为 0 的情况，影响最后概率的计算）。由于规律延续时间越长，在采样初得到的数据便越能够代表之后预测的情况，所以窗口的选择最好在参数估计的准确性（样本量越大越好）和规律的可延续性（估计的时间点与采样的初始点越近越好）取一个平衡。

模型有两条假设：①相对价格挡位上的订单的到达、取消及买单，卖单的成交遵循泊松过程；②订单相关事件的泊松率在短期内（从几分钟到数小时）可以视为不变。

直观上做出第 1 条假设的原因为，只要随机事件在不相交时间区间是独立发生的，而且在充分小的区间上最多只发生一次（如果我们把一单的数手分散在很短的时间里），它们的累计次数就是一个泊松过程，其次，泊松过程具有良好的可计算性质，适合于在高频下进行预测，可以快速地得到结果，为交易执行提供参考。

为了检验第 1 条假设，考察相邻两个事件（例如市价买卖单的到来，以及相对价格挡位上订单的到来和撤销）的间隔时间，观察其对数直方图是否为线性，因为对于服从泊松过程的随机事件，在时间$[t,t+\tau]$内发生的事件次数的概率分布为

$$P[(N(t+\tau)-N(t))=k]=\frac{e^{-\lambda\tau}(\lambda\tau)^k}{k!}\quad k=0,1,\cdots \tag{5-134}$$

令 $k=0$,则 $P[(N(t+\tau)-N(t))=0]=e^{-\lambda\tau}$,即这段时间间隔服从指数分布。对等式两边同时取对数,得到 $\log(P[(N(t+\tau)-N(t))=0])=-\lambda\tau$,即可证明,若订单的事件发生遵循泊松过程,则其相邻两事件间隔时间对数直方图应为线性。

为了检验第 2 条假设,即订单相关的泊松率在短期内(从几分钟到数小时)可以视为不变,统计某时刻前两小时(因为在接下来的回测中选用的窗口是两小时)在相对价格挡位上的各种事件和买卖市价单发生的次数,然后又用该时刻后的一小时在相对价格挡位上各种事件和买卖市价单发生的次数计算并代表该时刻各种事件的泊松率。

试图缩短泊松率的估计窗口以使该假设更加符合实际,然而由于在过短的时间里订单各种事件的频率无法准确地表征其真实的泊松率,经过试验选取估计窗口为两小时来表征其泊松率。得到在要检验的时刻前两小时与后一小时的事件发生次数之后,可以使用 Przyborowski-Wilenski 方法检验在各个相对价格挡位上事件的泊松率在 5% 显著水平上是否发生了变化。

限价订单簿的一个很重要的功能就是能提供用以预测未来短期的变量的信息,这些在实际交易执行中有帮助的变量包括中间价格上涨或者下跌的概率,中间价格变动之前的限价单被成交的概率,以及当买一价和卖一价只相距一个最小价格挡位时,限价单在对方报价反向移动之前被成交的概率。这些变量可以表示为基于当前订单簿的状态的这些事件的条件概率,而这些条件概率由于涉及递归导致的无限循环,所以无法得到显式解,如果用蒙特卡洛模拟进行数值求解,则需要进行大量模拟,在时间上不符合高频数据需要在极短时间(例如下一个订单到来之前)进行预测的要求,会出现等到订单已经到来之后才给出预测结果的情况。为了解决这一实际问题,使用 Abate 和 Whitt(1999)的方法,求得以连分数表示的上述事件对应的随机变量的 Laplace 变换 $\hat{f}$,并利用两个相互独立的随机变量的和的 Laplace 变换为这两个随机变量的 Laplace 变换的积的性质:$\hat{f}_{X+Y}(s)=E[e^{-s(X+Y)}]=E[e^{-sX}]E[e^{-sY}]=\hat{f}_X(s)\hat{f}_Y(s)$,得到对交易执行有帮助的随机变量的 Laplace 变换。

对于每个价格挡位上的订单数量,在满足之前两个假设的前提下,可以抽象为一个排队模型。对于一个排队模型(或称生灭过程),假设其到达率 λ 和在状态 $i\geqslant 1$ 的离开率 μ_i,记 σ_b 为从状态 b 开始第 1 次该过程到 0 所经过的时间。那么 $\sigma_b=\sigma_{b,b-1}+\sigma_{b-1,b-2}+\cdots+\sigma_{1,0}$,这里 $\sigma_{i,i-1}$ 表示的是排队过程从状态 i 第 1 次到状态 $i-1$ 所经过的时间,可以得知等式右边的每项均为相互独立的,我们记 $\hat{f}_b$ 为 σ_b 的 Laplace 变换,$\hat{f}_{i,i-1}$ 为 $\sigma_{i,i-1}$ 的 Laplace 变换,由 Laplace 变换的性质,即两个相互独立的随机变量的和的 Laplace 变换为这两个随机变量的 Laplace 变换的积的性质,我们有 $\hat{f}_b(s)=\prod_{i=1}^{b}\hat{f}_{i,i-1}(s)$。

为了计算下一次中间价格变动时,中间价格的上涨和下跌的概率,记 X_A 为最优卖价上的限价单数量,X_B 为最优买价上的限价单数量,$W_A(t)$ 为在最优卖价上剩余的限价单数

量，$W_B(t)$为在最优买价上剩余的限价单数量，ϵ_B 为 $W_B(t)$第 1 次到达 0 的时间(亦即最优买价上的订单全部要么被成交要么被撤销的时间)，ϵ_A 为 $W_A(t)$第 1 次到达 0 的时间(亦即最优卖价上的订单全部要么被成交要么被撤销的时间)，T 为中间价格第 1 次变化的时间。

那么，在给定 X_A 最优卖价上的限价单数量为 a，X_B 最优买价上的限价单数量为 b 的条件下，下一次中间价格变动为上涨的概率是 $P[p_M(T)>p_M(0)|X_A(0)=a, X_B(0)=b, p_S(0)=S]$，$S$ 为买卖价差。

那么对于在最优卖价上剩余的限价单数量 $W_A(t)$而言，增加一个订单的速率为之前提到的到达速率 $\lambda_A(S)$，后面的 S 代表这个速率与其距离最优买价的距离有关，减少一个订单的速率则为市场买单(成交价为最优卖价)到达速率 μ_A 加上在这个价格挡位和剩余限价单数量 n 上订单撤销的速率 $n\theta_A(S)$，共计 $\mu_A+n\theta_A(S)$。

对于在最优买价上剩余的限价单数量 $W_B(t)$而言，增加一个订单的速率为之前提到的到达速率 $\lambda_B(S)$，后面的 S 代表这个速率与其距离最优买价的距离有关，减少一个订单的速率则为市场卖单(成交价为最优买价)到达速率 μ_B 加上在这个价格挡位和剩余限价单数量 n 上订单撤销的速率 $n\theta_B(S)$，共计 $\mu_B+n\theta_B(S)$。

当买卖价差 S 仅为一个价格挡位的时候，我们想要知道的概率 $P[p_M(T)>p_M(0)|X_A(0)=a, X_B(0)=b, p_S(0)=S]$可以简化为最优卖单全部耗尽的时间 σ_A 小于最优买单全部耗尽的时间 σ_B 的概率，它的 Laplace 变换由 $\hat{f}_a^1(s)\hat{f}_b^1(-s)$给出，这里的 s 表示任意复数，不是之前的买卖价差 S。

而对于买卖价差 S 大于或等于两个价格挡位的时候，记 σ_A^i 为距离最优买价 i 个挡位第 1 次有订单到来的时间，σ_B^i 为距离最优卖价 i 个挡位第 1 次有订单到来的时间，那么此时中间价格第 1 次变化的时间 T 是 σ_A，σ_B，以及中间某挡位最快的订单到来时间的最小值记作 $T=\sigma_A\wedge\sigma_B\wedge\min\{\sigma_A^i,\sigma_B^i,i=1,\cdots,S-1\}$。这些 $\{\sigma_A^i,\sigma_B^i\}$(其中 $i=1,\cdots,S-1$)的到来速率均可由之前的参数估计部分得到。那么中间价格向上运动的概率为最优卖单全部耗尽与中间挡位出现买单更快者小于最优买单全部耗尽与中间挡位出现卖单更快者的概率记为 $P[\underline{\sigma_A}\wedge\sigma_B^1\wedge\cdots\wedge\sigma_B^{S-1}<\underline{\sigma_B}\wedge\sigma_A^1\wedge\cdots\wedge\sigma_A^{S-1}]=P[\sigma_A\wedge\sigma_B^{\Sigma_B}<\sigma_B\wedge\sigma_A^{\Sigma_A}]$，其中 $\sigma_B^1\wedge\cdots\wedge\sigma_B^{S-1}=\sigma_B^{\Sigma_B}$，$\sigma_A^1\wedge\cdots\wedge\sigma_A^{S-1}=\sigma_A^{\Sigma_B}$，符号 $\wedge$ 表示取两数中之更小者。

随机变量 $\sigma_A\wedge\sigma_B^1\wedge\cdots\wedge\sigma_B^{S-1}-\sigma_B\wedge\sigma_A^1\wedge\cdots\wedge\sigma_A^{S-1}$ 的 Laplace 变换可以通过以下公式求得

$$\hat{F}_{a,b}^S(s)=\frac{1}{s}\left(\hat{f}_a^S(\Lambda_{B,S}+s)+\frac{\Lambda_{B,S}}{\Lambda_{B,S}+s}(1-\hat{f}_a^S(\Lambda_{B,S}+s))\right)\cdot$$
$$\left(\hat{f}_b^S(\Lambda_{A,S}-s)+\frac{\Lambda_{A,S}}{\Lambda_{A,S}-s}(1-\hat{f}_b^S(\Lambda_{A,S}-s))\right) \tag{5-135}$$

其中，$\Lambda_{B,S}\equiv\sum_{i=1}^{S-1}\lambda_B(i)$，$\Lambda_{A,S}\equiv\sum_{i=1}^{S-1}\lambda_A(i)$，代表中间挡位出现买单/卖单的速率。$\hat{f}_j^S(s)=$

$\left(-\frac{1}{\lambda(S)}\right)^{j}\left(\prod_{i=1}^{j}\Phi_{k=i}^{\infty}\frac{-\lambda(S)(\mu+k\theta(S))}{\lambda(S)+\mu+k\theta(S)+s}\right)$，这个由上面的排队过程给出，表示在买卖价差 S 时，在最优买价或者卖价上所有订单要么被成交要么被取消所用去的时间，即上面的最优卖单全部耗尽/最优买单全部耗尽所用去的时间。

式中的连分数按照如下方法计算可得，我们记连分数 $w\equiv\Phi_{n=1}^{\infty}\frac{a_n}{b_n}$，其中部分分子序列 $\{a_n,n\geqslant 1\}$ 和部分分母序列 $\{b_n,n\geqslant 1\}$ 为复数且 $a_n\neq 0$，$w_n=t_1\circ t_2\circ\cdots\circ t_n(0)$，$n\geqslant 1$，$\circ$ 为复合运算符，如果 $w\equiv\lim_{n\to\infty}w_n$，则 w 即为连分数的值。这个连续近似可以由欧拉于 1737 年提出的递归方法计算，$w_n=\frac{P_n}{Q_n}$，其中 $P_0=0$，$P_1=a_1$，$Q_0=1$，$Q_1=b_1$，

$P_n=b_nP_{n-1}+a_nP_{n-2}$

$Q_n=b_nQ_{n-1}+a_nQ_{n-2}$。

在套利过程中，下一个限价单往往比下市价单去主动成交一个对手最优报价能获得更好的价格，然而这会使交易员面临着这个限价单未必成交的风险。下市价单能确保订单被执行，而限价单则停留在订单簿上直到要么匹配的订单把它成交掉，要么被取消掉。那么限价单在中间价格移动之前被成交的概率对于决策究竟是下市价单还是下限价订单就有很好的量化参考意义。特别是，当买一价和卖一价只相距一个最小价格挡位时，这个概率即是限价单在对方报价反向移动之前被成交的概率。

记 $NC_b(NC_a)$ 为在时间点 0 下不撤回的买一价订单(卖一价订单)这一事件(这即为对自己下的限价单的假设，在中间价格变动之前不打算撤销该订单)。那么，在中间价格变化之前，这个订单被执行的概率为 $P[\epsilon_B<T\mid X_B(0)=b,X_A(0)=a,p_S(0)=S,NC_b]$，其中 ϵ_B 为订单被执行的时间，T 为中间价格第 1 次变化的时间。自己下的限价单与市场其他交易员下的限价单的区别在于，自己下的订单的撤销是一个确定性的事件，可以自行决定什么时候撤单，所以不再随机，由于相同价格时间优先的原则，后面到来的同一相对价格挡位的订单与我们的订单是否被成交不再有关，无论这些订单是到来还是到来后又撤销，当然，后面到来的相同挡位的订单是不可能在比我们成交之前成交的，因此，之前的模型需要把自己下单的相对价格挡位的到达率设置为 0。自己下的订单本身也无须考虑随机的撤单率，因为这对于自身是一个确定的事件，想要计算的是在中间价变动之前限价单一直挂在那里，然后被其他交易员成交掉的概率。

$$\hat{F}_{a,b}^{S}(s)=\frac{1}{s}\hat{g}_b^S(s)\left(\hat{f}_a^S(2\Lambda_S-s)+\frac{2\Lambda_S}{2\Lambda_S-s}(1-\hat{f}_a^S(2\Lambda_S-s))\right)\tag{5-136}$$

其中，$\hat{g}_b^S(s)=\prod_{i=1}^{b}\frac{\mu+\theta(S)(i-1)}{\mu+\theta(S)(i-1)+s}$，相应地，如果要计算自己在卖一价挡位的订单的成交可能性，则只需将公式中 a 与 b 的位置进行互换。

在得到我们想知道的随机变量的 Laplace 变换，包括最优卖单全部耗尽的时间减去最优买单全部耗尽的时间及最优卖单全部耗尽与中间挡位出现买单更快者所用时间减去最优

买单全部耗尽与中间挡位出现卖单更快者所用时间之后，把它除以 s 以后逆变换回去可以得到这些随机变量的累积分布函数，然后计算它们在 0 点的取值即可得到接下来想要计算的两种概率，中间价格涨跌概率及中间价格变动之前在买一或者卖一的订单被成交的概率。

需要用到 Laplace 逆变换，可由以下公式近似，将之前得到的 Laplace 变换 $\hat{F}_{a,b}^{S}$ 代入式中的 $\hat{f}$，经验上选取 $n=15$，$m=11$，然后通过欧拉和就可以得到 Laplace 数值逆变换。

$$E(m,n,t)=\sum_{k=0}^{m}\binom{m}{k}2^{-m}s_{n+k}(t),\quad s_n(t)=\frac{e^{A/2}}{2t}Re(\hat{f})\left(\frac{A}{2t}\right)+\frac{e^{A/2}}{t}\sum_{k=1}^{n}(-1)^k a_k(t) \tag{5-137}$$

其中，$a_k(t)=Re(\hat{f})\left(\frac{A+2k\pi i}{2t}\right)$。

5.6.4　泊松过程模型的应用效果

将模型运用于实际行情，以 FR007.5Y.IRS（5 年期挂钩 FR007 的利率互换）的固定利率行情为例，查看其在 2019 年 7 月 1 日至 31 日的交易日的预测表现。对于中间价下一次变动为上涨的概率，结果如图 5-10 所示。

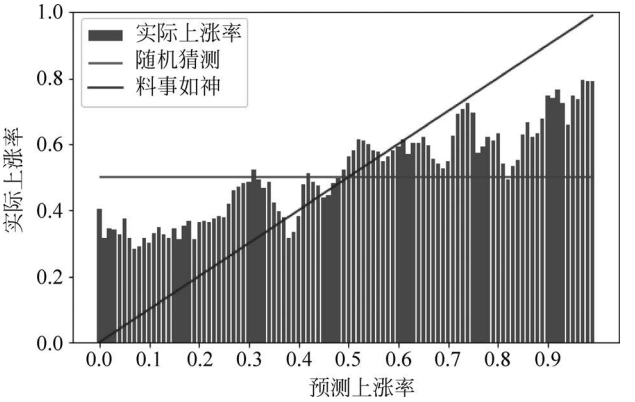

图 5-10　上涨率预测表现

如图 5-10 所示，模型对于预测上涨率有明显效果，比随机猜测要好很多。例如对于模型预测上涨率为 0.8 时的情况，实际的上涨率确实为 0.8，模型预测上涨率为 0.2 的部分，实际的上涨率为 0.3。对于预测上涨率，在极端情况下，例如预测上涨率大于 0.85 或小于 0.1 时，模型的准确率虽然大方向是对的，比随机猜测准很多，但准确率距离完美预测还有一定距离，其中的不足我们会设法在改进时进行弥补，但从效果来看，显然这个模型能够对交易员的交易优化起到辅助作用。回到一开始的例子，如果交易员准备建仓而模型认为上涨率为 0.8，则最终实际上涨率应有 8 成，交易员立即下市价单，以及时抓住交易机会，避免限价单最终很有可能在中间价格移动之前未被成交。

对于下一次中间价格变动是下跌的概率，如图 5-11 所示。

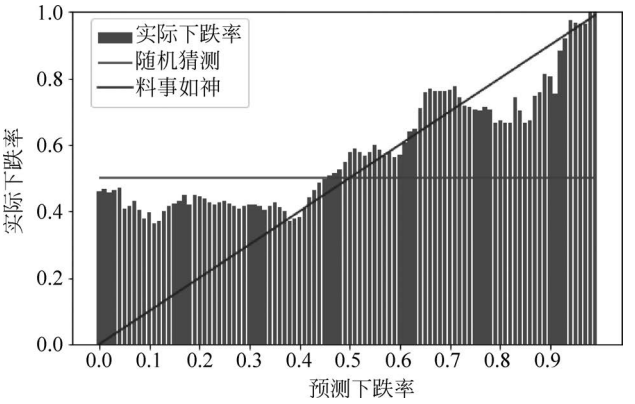

图 5-11 下跌率预测表现

模型在预测下跌率时，在预测值大于 0.5 时尤为准确，例如当预测下跌率为 0.9 时，如果交易员准备建仓持有多头头寸，则限价单是一个好的选择，因为中间价格很有可能会滑落，限价单很有可能会被成交，那么就在没有冒很大成交不确定性风险的情况下，获得了更优的价格，降低了成本，也就增加了利润。在小于 0.5 时虽然比随机猜测好，但是离实际下跌率还有一定距离。这是由模型的特质决定的。当订单的到来快于订单的成交和撤销时，模型会认为排队过程的首达时间为无穷大，此时预测的上涨概率或者下跌概率会坍缩为 0 附近的数，但事实上，这种情况并不会延续，也就是第 2 条假设被违反了。第 2 条假设要求我们在相对价格挡位上的泊松率保持稳定，也就是可以通过前几小时的泊松率近似替代当前到下一次中间价格变化之间的泊松率。这个假设在之前的假设检验部分被发现部分违反。这可能导致两种情况，一是我们的模型计算出来的上涨率/下跌率不准确，因为输入的过去数小时的泊松率不能准确地代表当前的泊松率；二是计算出来的上涨率/下跌率不符合逻辑，例如远大于 1 或远小于 0，这种情况多见于在过去几小时内，泊松率估计采样过程中，订单的到来快于订单的成交和撤回，导致模型认为排队过程永远不会结束，无法比较两个排队首达时间的大小，从而输出异常值，其中这种情况虽无法准确地预测其下一次中间价格变化的涨跌概率，却对于交易员执行交易十分有利，因为这意味着当前价格挡位上积累了一些订单，交易员不用担心中间价格的立即变化，除非成交速率或撤销速率突增。

5.7 订单簿信息作为交易信号

在之前的研究中，以交易中对于短期内中间价格预判的需求为导向，根据五档订单簿及其历史行情，构建了一个预测短期中间价格走势的模型。

接下来，将这个模型投入实战，通过进行更长时间的持仓策略进行回测，考察其对于实际交易的意义。

在多个品种上，当前策略的回测效果良好，收益可观。下面简要地介绍策略逻辑及其金融含义，以及数量型参数选择的依据，其中回测时每次交易数量相等，下文中的成交速率即

为5.6节中的市场单的到达速率，下文中的中间价涨跌的可能性即为5.6节中的预测上涨率或者预测下跌率。我们以ETF的日内交易为例，如159755.SZ电池ETF和512000.SH券商ETF。

5.7.1　开仓逻辑

当无持仓时，若卖价上成交速率/买价上成交速率＞23且买价上成交速率＞1且卖一量稀薄(可选)时，开多仓。

当无持仓时，若买价上成交速率/卖价上成交速率＞23且卖价上成交速率＞1且买一量稀薄(可选)时，开空仓。

经济含义为卖价上成交速率/买价上成交速率＞23，这是策略做对方向开对仓位的核心。之前提到的涨跌概率的输入变量之一就是卖价上成交速率与买价上成交速率。这个比例能有效地衡量市场上的买卖力量的强弱。之所以不选用盘口挂单量，是因为在实盘中时而会观察到“压单”现象，即盘口某个方向挂单很多，但市场却朝着此方向的反向运行，而成交相对来讲是一个更加真实的度量，因为交易双方需要付出真金白银。如果卖价上成交速率/买价上成交速率非常极端，则非常有可能市场上已经出现了不均衡的现象。这种不均衡对价格运动方向的指征通常会持续相当长的一段时间，比模型输出的涨跌概率的有效时间更长(模型输出的涨跌概率在理论上的有效期即截至下一次中间价格的变化时间点)。例如在交易国债期货的过程中，当2021年7月7日下午2时许，市场出现大量成交性质为“多开”的交易，压倒性超过其他成交性质的交易。反映到卖价上成交速率/买价上成交速率就是这个指标很大，往往意味着买的力量非常强势。全天十年期国债期货增仓1万4千余手，在卖价上成交的比例很大。当天国债期货午盘后大幅收涨。事实证明，当天晚上7时许即传来降低存款准备金率的消息。这一例子即是卖价上成交速率/买价上成交速率这一指标作用的体现。

买价上成交速率＞1这一条件是为了防止在成交不活跃的时候出现对价格走势的误判。例如买价上成交速率为0时会非常容易地触发上面的条件1，但是这并不属于要刻画的价格变化的征兆，所以设置阈值排除此情况，而且实际上，价格的变化与成交量是高度相关的，基本面的改变通常会导致大量的换手及价格的大幅波动，因此为了使开平仓有利可图，希望通过量的放大来捕捉价格波动的放大。

卖一量稀薄(可选)这一条件是为了提高模型的胜率并降低磨损，同时充分发挥算法交易在抢单中的优势。在使用中间价格涨跌预测概率模型盯盘的过程中，发现当模型给出中间价格的方向的对盘价上的量很稀薄时，往往胜率较高。这是因为模型对于泊松率的假设是在短时间不变。如果对盘价上的量太厚，而在一段时间后相对价格挡位上的泊松率发生变化，则击穿对盘价的可能性就大大降低。如果没有击穿对盘价且与本方同向的其他市场参与者的订单没有向期望的价格运动，则在平仓时就需要多付出1个或更多tick，所以设置此条件排除对盘价上的量很厚的情形。如果交易频率较低，则敞口较大，可以去除此条件，但如果交易频率较高，则最好加上此条件。

5.7.2 平仓逻辑

当持有多头的时候,如果中间价跌可能性>0.6,则平多头。

当持有空头的时候,如果中间价涨可能性>0.6,则平空头。

当日终时,如果持有多头,则平掉多头,如果持有空头,则平掉空头。

经济含义为中间价跌可能性>0.6,这一信号从交易执行的层面来讲,可以使获得的平仓价格更优。从长周期的层面来讲,一般只有成交较为活跃,并且短期趋势确实与之前的建仓方向相反时才会发出这个信号。在开仓逻辑后,当我们开仓方向与大趋势一致时,由于订单的成交、到达和撤回会将价格向与我们开仓方向相同的方向推动,这个信号在顺势的时候是不容易被触发的,有助于让顺趋势的利润奔跑。

5.7.3 回测结果

截至2021年11月,回测取得了不错的收益率,胜率和盈亏比都有着不错的表现,见表5-1。

表5-1 在ETF上的订单簿策略回测统计

统计量名称\代码	SZ159755	SH512000
回测天数(交易日)	97	209
累计收益/元	990 000	450 000
年化收益率/%	19.99	5.11
最大回撤/元	80 000	190 000
胜率/%	73.9	50.0
盈亏比	2.408	1.6781
总交易次数	51	36
底仓占资/元	12 867 220	10 612 710

5.7.4 订单流与消息面的关系

当没有公开信息出现时发生的急涨或者急跌,趋势会继续延续,适合做趋势策略;当出现公开信息后,趋势会被信息劣势,但定价能力弱的大众推波助澜后反转,此时适合做反转策略,所以后期是否会发生反转,关键看基本面的变化量是否已经被反映在价格的变化量中。对于那些持有私有信息的市场参与者而言,他们有私有信息,同时有相对于大众更高的估值能力,因此正是他们快速买入导致价格上涨逐渐收敛至基本面变化后的公允价格附近。为了获利,这部分私有信息持有者并不会显著地在高于其认为的基本面变化后的公允价格以上进行买入,所以买入的终点在公允价格附近,并且通常不会由这部分私有信息持有者推高至公允价格以上。此时又会吸引一批趋势跟踪者买入,然而私有信息持有者希望等待到价格更高、交易对手所能提供的成交量更大的价格卖出。一旦利多的新闻发布,接受公开信息的大众会开始关注到这一被利多的资产并开始买入,然而缺乏定价能力使最新价格逐步地超过公允价格。这时资产被不断地换手,价格继续被推高,直到出现拐点,此时新增的预

期价格上涨的交易者将减少，而一开始的私有信息持有者逐步地将持有的存货抛出，供需失衡且供大于求，这就导致了趋势的反转。趋势反转的终结点通常在公开信息发布时的价格之下，在趋势开始时的价格以上。

以2022年11月4日的股票市场为例，开盘第一分钟，沪深300股指期货即拉涨0.55%，此后分钟线5连阳，与此同时，港股也快速拉升，上涨幅度令人瞠目，此时还没有任何公开消息，那么如果趋势在延续，则说明私有信息的持有者仍然认为公允价格在当前的价格之上，此时简单采取趋势策略即可，截至午盘，沪深300股指期货涨3.04%。中午时分，有消息称中国防疫政策可能于近期发生重大改变。下午还没有开盘，这一消息就得到了广泛传播，有人相信，有人怀疑，但毫无疑问，有大量得到公开信息的交易者准备买入。下午1点刚开盘，期货跳空高开并收一根0.45%的分钟级别大阳线，此时各个ETF卖盘早已躲到价格较高处。随后在午盘的基础上沪深300股指期货放量上涨0.94%，然后逐步开始回落（此时信息已公开且涨势明显放缓，当获利回吐超过一定比例，如20%时即可止盈），并两次在下午一点零一分这根K线的上沿获得支撑，最后当日收盘于这根K线上沿，并于此后的4个交易日内回落至2022年11月4日的早晨开盘第一根K线上沿。策略的盈利验证了前述价格和订单趋势与基本面关系理论的正确性。

5.8　订单簿的机器学习模型

在之前的模型中都是通过对于订单簿的特征，如静态特征的挂单量，动态特征的订单新增、撤回、成交等信息，然后通过推理演绎的办法去对后市进行预测，然而随着当今算力的提高，新模型的提出和数据科学的发展，直接以高频海量数据为基础的机器学习订单簿模型也逐步投入业界应用。订单簿作为有着高维和长时间序列的数据，特别适合于使用机器学习的办法去发现其中的规律。尽管由于金融数据具有信噪比较低的特点，运用于预测非常容易产生过拟合的结果，但通过合适的模型选择和适当的训练，依然能从其中发现那些非线性、不直观的规律。同时随着学术界和业界对机器学习方法的研究，模型也不再以“黑盒”的方式工作，而是能够被研究员更清晰地解读。

5.8.1　订单簿的DeepLOB模型

根据Zihao Zhang等的研究，通过卷积神经网络（Convolutional Neural Network，CNN）和长短期记忆网络（Long Short-Term Memory，LSTM）模型，对于订单簿进行建模。订单簿的数据与大多数金融数据一样，长期是非平稳的。对于订单簿的后几挡尤其如此，因为后几档的订单经常因为对于后市行情的预期进行增减，以及报撤单。使用海量数据有助于构建通用模型对各个品种进行预测，而无须大幅度修改模型的结构。

在这个模型中通过Inception Module去构造特征，从而推断不同时间戳的行情之间的相互作用。之前的章节我们都是通过提取订单的到达、撤回、成交这些有直观的实际意义的特征来对订单簿的动态进行推断，而Inception Module处理后的特征图会被传入LSTM，从

而捕捉动态时序特征。CNN 则能很好地通过过滤器进行信息提取。LSTM 则能避免通常的循环神经网络中常见的梯度消失问题。Inception Module 能用来推断最优的提取特征的衰减率。

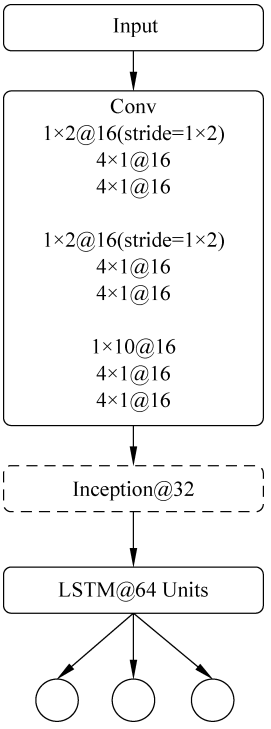

图 5-12 DeepLOB 模型的整体结构

模型输入的是不包含集合竞价的连续竞价交易时间段的订单簿,也就是买单价 $P_b(t)$ 和买单量 $V_b(t)$,以及卖单价 $P_a(t)$ 和卖单量 $V_a(t)$,其中 $P(t)$和 $V(t)$都是表征订单簿各个挡位的量价的向量。输入可以表示为 $\boldsymbol{X}=[x_1,x_2,\cdots,x_t,\cdots,x_{100}]^{\mathrm{T}}\in\mathbf{R}^{100\times 400}$,这里的 $x_t=[p_a^{(i)}(t),v_a^{(i)}(t),p_b^{(i)}(t),v_b^{(i)}(t)]_{i=1}^{n=10}$,其中 $p^{(i)}$ 及 $v^{(i)}$ 表示的是第 i 挡位的订单簿价、量信息。根据历史文献来看,最优挡位一般包含 8 成以上的信息。Zihao Zhang 等的 DeepLOB 模型的结构如图 5-12 所示。

如图 5-12 中的 1×2@16 表示的是一个有着 16 个(1×2)过滤器的卷积层,其中第一维沿着时间戳做卷积,而第二维沿着不同的订单簿挡位做卷积。Input 表示输入。那么第 1 层卷积层(Conv)就归纳了每一档的量价对的信息,这里的步幅(stride)等于 2,在此也非常必要,因为卷积层的参数是共享的,从逻辑上来讲,我们不能对量价对和价量对施加相同的参数。第 1 层卷积层只提取每一挡位的信息,为了汇聚不同挡位的订单簿信息,我们使用步幅为 1×2 的 1×2 过滤器,然后我们使用 2 层的卷积层得到了(100×10)的特征图,将一个 1×10 的过滤器作用于这个特征图,我们将得到(100×1)的特征图。

在这个过程中,每个卷积层面都进行了 0 填充以确保特征的维度不变。同时激活函数采用的是 Leaky-ReLU,超参数是 0.01,这是通过对于验证集的网格搜索来得到的最优超参数。

仅仅在 Inception Module 以内使用了池化层,因为尽管池化层能帮助我们提取信息,但是它的平滑的特性容易导致欠拟合。在时间序列中,特征的位置是非常重要的,所以没有大量地采用池化层,如图 5-13 所示。

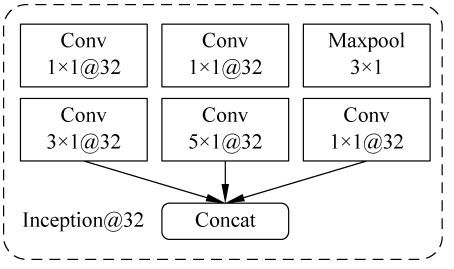

图 5-13 模型中使用的 Inception Module(图片来源参考文献)

所有的标准卷积层都有着相同的大小，例如如果使用(4×1)大小的过滤器，我们就是在捕捉4个时间戳之内的相互作用，然而，可以通过Inception Module去抓取更多时间戳的订单簿动力学特征，其中的最大池化层(Maxpool)步幅为1，并且用0填充。Inception@32表示的是这里面的所有卷积层都有32个过滤器。图5-13中的1×1小型网络中网络可以捕捉数据中的非线性关系。

在做完以上步骤之后，模型产生了大量的特征，为了提取其中的时间序列关系，使用LSTM单元代替全连接层，以降低参数的个数，避免过拟合。64个LSTM单元引入了约60 000个参数。最后一层是Softmax层，用以预测出与输出状态对应的概率。

5.8.2　模型的应用

在A股市场上，订单簿在Level-1数据中通常以数据切片的形式给出，不同的交易所开始推送的时间不同，但通常上海证券交易所和深圳证券交易所的数据切片的间隔是3s，其中上海证券交易所的连续竞价交易时间每天共4h，对应的是4800个切片。我们使用历史的数据特征对其进行标准化。

取过去的一个月中的每个特征的均值和方差作为标准化的参数，然后作用于当前交易日，这样既不会用到未来信息，又能使输入特征处于同一尺度，让不同维度之间的特征在数值上有一定的比较性，可以大大地提高分类器的准确性。

在数据的预处理方面，由于市场的涨跌和持平的样本分布是不均匀的，例如在预测高频价格序列的涨跌的过程中，如果在大部分情况下行情寡淡，模型就会在行情寡淡的时刻进行充分拟合，而对那些市场大幅波动但对于盈利非常关键的波动则欠拟合。为了避免这种情况的发生，可以将那些样本数量较少但是对于盈利非常关键的高波动时刻通过train_on_batch函数中的class_weight参数赋予较高的权重，使模型能更好地拟合对于盈利关键的部分。在当前模型中等权进行训练的表现已经足够好。

在模型拟合方面，通常情况下简单地将数据分为训练集、验证集、测试集3方面，但效果并不好。可能的原因是：A股市场在交易过程中时常围绕着一个主线展开行情，投资者的行为很大程度上决定了市场的范式，因此如果不采用滚动训练的方法，而是试图使用前10个月数据拟合出来的模型去预测后1个月乃至后2个月的行情，没有充分地用到最新的信息，所以我们采用滚动的训练方法。

首先对于特征进行标准化，特征包括从买一到买五，以及从卖一到卖五的量价对，其中量价按照顺序排好以确保神经网络共享权重的正确性，其余特征包括前述章节提到的订单的到达、撤回、成交速率及其衍生出来的变量，然后用前10天的每个特征的标准差和均值作为标准化的依据，对新的一天的特征进行标准化。这样既不会引入未来信息，又运用了能使用的最新的信息，因为市场的量价情况会随着时间演变，因此做滚动的标准化以确保模型的输入被合理地标准化了。预测目标值的生成则是3个数据帧以后的close价格的相对状态，即上涨、持平、下跌。在本例中是9s以后的相对状态，这是一个较高频的预测，可以用于做市商报价及交易执行。

由于运算量较大,数据(包括训练数据、模型参数)占用的内存也较多,大型机器学习模型一般使用 GPU 进行训练。本例中,训练使用 NVIDIA RTX A6000 显卡。在普通的台式机中,内存难以一次存下海量的数据进行运算。同时,运算速度也是重要的考量,对于下一交易日就需要使用的高频模型而言,训练时间必须小于前日收盘到次日开盘之间的时间间隔。

训练过程同样也是滚动的。在训练神经网络的过程中,我们使用前 30 个交易日的数据作为训练集,并令 epochs=90 来训练这个模型,经验法则中 epochs 通常是特征数量的 3 倍左右。每次模型只运用于下一个交易日的预测,对于再下一日的预测,则需要重新滚动训练。

5.8.3 模型的效果

对于用于金融交易的模型而言,通常情况下,预测结果的准确率重要于召回率,因为一旦模型预测后续的价格波动引起开仓,就会引起策略盈利的波动,如果准确率低而召回率高,则过度的错误交易就会大幅地侵蚀利润,其效果往往不如"三年不开张,开张吃三年"的高准确率而低召回率的模型。对于高频策略而言,其摩擦成本往往相对于盈利来讲是巨大的,因此在模型评价中,对于准确率的重视通常高于召回率。

根据模型的预测结果得到如图 5-14 所示的混淆矩阵,其中横轴是预测值,纵轴是实际值,每个对应的小方块中的数字是其对应的样本个数。例如左上角的 1 579 806,指的是共有 1 579 806 个样本在模型中 3 个数据帧后 close 价格被预测(predicted)为下跌(down)而实际(actual)确实发生下跌现象。

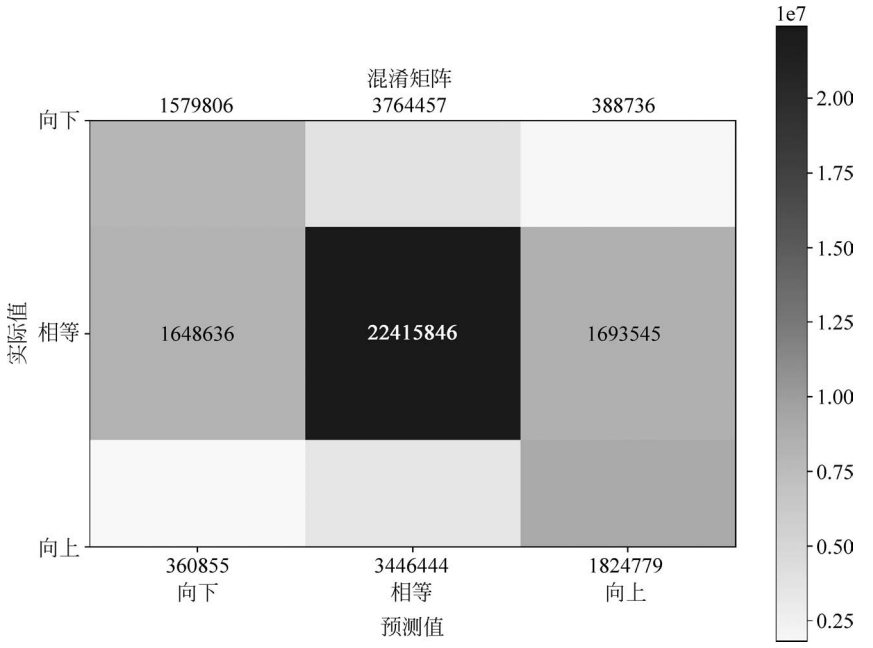

图 5-14 DeepLOB 模型的输出结果之混淆矩阵(图片来源参考文献)

如图 5-14 所示，是将上述模型应用于 515220.SH 煤炭 ETF 上的结果，可以看出，斜对角线（左上到右下）上的元素颜色是较深的，说明当模型预测是上涨的时候，接下来的 3 个数据帧后，大部分情况是上涨或者至少是持平的，只有极少数情况是下跌的；当模型预测是下跌的时候，接下来的 3 个数据帧之后，大部分情况是下跌或者至少是持平的，只有极少数是上涨的；当模型预测是持平的时候，大部分情况就是持平的。DeepLOB 模型的预测对于在准备建仓时订单类型的选择乃至一些短期策略的构建是大有裨益的。

5.9　强化学习

5.9.1　强化学习简介

强化学习（Reinforcement Learning，RL）属于一种人工智能，又称增强学习，是机器学习的范式和方法论之一，用于描述和解决智能体（Agent，或称为决策者）在与环境的交互过程中通过学习策略以达成回报最大化或实现特定目标的问题。

人工智能领域中有许多类似的趋利避害的问题。例如，当人们要精进围棋的下法时会去阅读棋谱，而这些棋谱是通过历史上无数围棋大师通过无数盘对弈总结出来的经验。著名的围棋 AI 程序 AlphaGo 可以根据不同的围棋局势进行审时度势（观测环境），然后做出最优决策（动作）。如果它决策合理，它就会赢，如果决策失误，它就会输，并从对弈过程中学习到有用经验。得益于计算机高速运行，它可以比人类以更快的速度获取下棋的经验，从而不断地提高自己的棋艺，这就和行为心理学中的情况如出一辙，所以人工智能借用了行为心理学的这一概念，把与环境交互中趋利避害的学习过程称为强化学习。

强化学习是一类特定的机器学习问题。在一个强化学习系统中，决策者可以观察环境，并根据观测做出行动。在行动之后，能够获得奖励。强化学习通过与环境的交互来学习如何最大化奖励。例如，一个机器人走迷宫的问题，机器人观察周围的环境，并且根据观测来决定移动方向。错误的移动会让机器人浪费宝贵的时间和能量，正确的移动会让机器人成功地走出迷宫。在这个例子中，机器人的移动就是它根据观测而采取的行动，浪费的时间、能量和走出迷宫的成功就是给机器人的奖励。强化学习正在改变人类社会的方方面面：基于强化学习的控制算法已经运用于机器人、无人机等设备，基于强化学习的交易算法已经部署在金融平台上并取得了超额收益。由于同一套强化学习代码在使用同一套参数的情境下能解决多个看起来毫无关联的问题，所以强化学习常被认为是迈向通用人工智能的重要途径。

如果读者已经对数学基础和机器学习的基本概念有了一定的了解，则可以跳过阅读本节。如果读者对上述内容不太了解，则要认真学习这一部分知识。

5.9.2　强化学习基本概念

强化学习的正式介绍以网格世界这个环境来做例子，如图 5-15 所示。

在图 5-15 的网格世界中，有一个机器人在里面游走，在这个网格世界中，有的网格是可进入的，而有的网格是禁止进入的，右下角那个网格则是目标网格，机器人走进去就会得到

奖励。机器人只能上下左右移动，不能斜着移动。决策者的目标是探索最优路径，从而到达目标网格，并且尽可能地不要进禁止区域，同时也要避免走出网格世界的边界，例如在网格世界左上角的格子的位置最好不要向上走或者向左走。

这个机器人强化学习模型中的智能体，网格世界则是环境。智能体与环境的交互如图 5-16 所示。

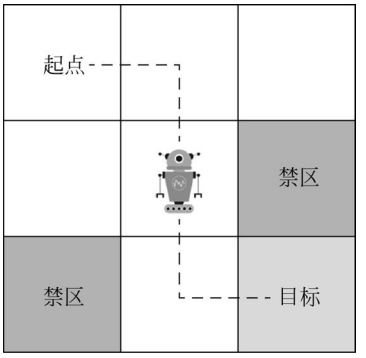

图 5-15　网格世界

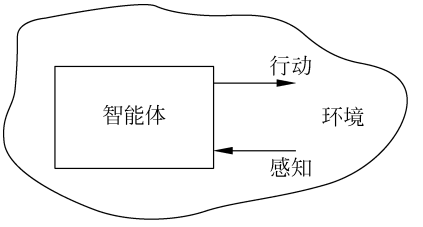

图 5-16　人工智能体

读者需要关心如下几个重要概念。

(1) State(状态)：描述相对于环境的状态，在这个网格世界中一种状态指的是一个位置(一个具体的网格位置，如图 5-17 所示)。

(2) 状态空间：是一个集合$\mathcal{S}=\{s_i\}_{i=1}^{9}$。

(3) Action(行动)：a_1 表示向上移动、a_2 表示向右移动、a_3 表示向下移动、a_4 表示向左移动、a_5 表示原地不动，如图 5-17 所示。

(4) 行动状态空间：是一个集合$\mathcal{A}=\{s_i\}_{i=1}^{5}$。

(5) 状态转移：$s_1 \xrightarrow{a_2} s_2$ 表示从状态 s_1 通过行动 a_2 到达状态 s_2。

注意，状态转移是有限制的，要考虑到状态转移的可行性，例如在这个网格世界中想要从只能到达状态 s_1 通过行动 a_1 到达状态 s_2 是不可能的，它会撞墙之后停留在状态 s_1 即 $s_1 \xrightarrow{a_1} s_1$。禁止区域也是有限制的，如 $s_5 \xrightarrow{a_2} s_5$，这是因为撞上了禁止区域 s_6 后被弹回原来的区域 s_5。

网格世界行动标识图如图 5-18 所示。

s_1	s_2	s_3
s_4	s_5	s_6
s_7	s_8	s_9

图 5-17　网格世界状态标识图

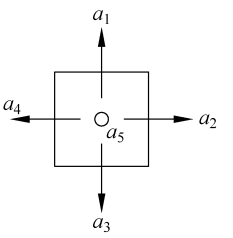

图 5-18　网格世界行动标识图

状态转移可以用状态表格来表示，见表 5-2。

表 5-2　状态转移表

状　态	行　动				
	a_1	a_2	a_3	a_4	a_5
s_1	s_1	s_2	s_4	s_1	s_1
s_2	s_2	s_3	s_5	s_1	s_2
s_3	s_3	s_3	s_6	s_2	s_3
s_4	s_1	s_5	s_7	s_4	s_4
s_5	s_2	s_6	s_8	s_4	s_5
s_6	s_3	s_6	s_9	s_5	s_6
s_7	s_4	s_8	s_7	s_7	s_7
s_8	s_5	s_9	s_8	s_7	s_8
s_9	s_6	s_9	s_9	s_8	s_9

虽然表格的形式比较直观，它的所有状态转移都是确定性的，但在实际应用场景中在大多数情况下存在不确定性的状态转移。

在网格世界中，s_1 有可能通过 a_2 到达 s_2，也有可能通过 a_3 到达 s_4。这里引入条件概率来表示行动状态转移概率。用 $p(s_2|s_1,a_2)=1$ 来表示从 s_1 通过 a_2 到达 s_2 的概率为 1。反之 $p(s_i|s_1,a_2)=0\ \forall i\neq 2$。条件概率能用来描述带有随机性的状态转移。假设这个网格世界中带有从上而下的风，从 s_1 通过 a_2 到达 s_2，也有可能被风吹到 s_5。这就赋予了状态转移成功到达某些位置的概率，在随机的情况下用条件概率的形式表达状态转移。接下来引入一个非常重要的概念——Policy(策略)。

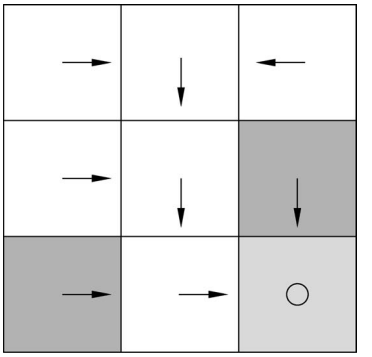

图 5-19　网格世界策略示意图

策略在网格世界中的直观理解如图 5-19 所示。

在图 5-19 中，一共有 9 种状态，每种状态对应一个行动，用箭头和圆圈表示。基于这个策略，可以得到一些路径，如图 5-20 所示。

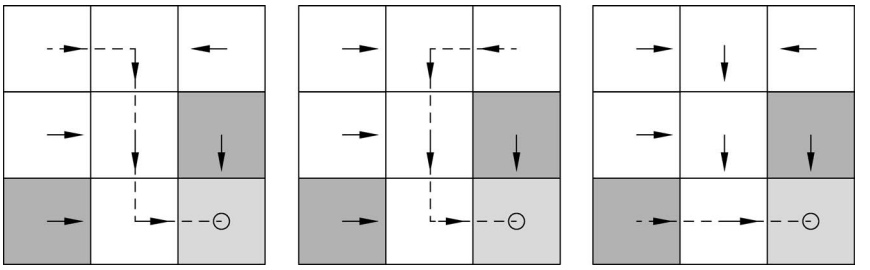

图 5-20　网格世界策略路径图

在做强化学习模型中，一般使用数学符号 π 来表示策略(一般在数学上 π 用来表示圆周率，在强化学习模型中用于表示策略)，策略是一个条件概率，如图 5-20 所示的策略中可以

使用 $\pi(a_1|s_1)=0$ 来表示在状态 s_1 时不采取 a_1 行动。使用 $\pi(a_2|s_1)=1$ 来表示在状态 s_1 时以百分之百的概率采取 a_2 行动。可以用一组 π 来表示一种状态(如 s_1)下的行动：

$$\begin{aligned}&\pi(a_1 \mid s_1)=0\\&\pi(a_2 \mid s_1)=1\\&\pi(a_3 \mid s_1)=0\\&\pi(a_4 \mid s_1)=0\\&\pi(a_5 \mid s_1)=0\end{aligned} \tag{5-138}$$

注意，针对一种状态下的采取所有行动的概率之和为 1，即

$$\sum_{a_i \in \mathcal{A}} \pi(a_i \mid s_n)=1 \tag{5-139}$$

策略也可以用表格形式来表达，读者可以试着把表 5-2 改为策略表，状态转移表通常记录了在每个状态下执行每个动作后，转移到下一个状态的概率。而策略表描述了在每个状态下，应当执行哪些行动的策略。

如图 5-20 所示的策略是一个确定性策略，在一个特定状态中要么百分之百地采取某行动，要么就不采取某行动，带有随机性的策略如图 5-21 所示。

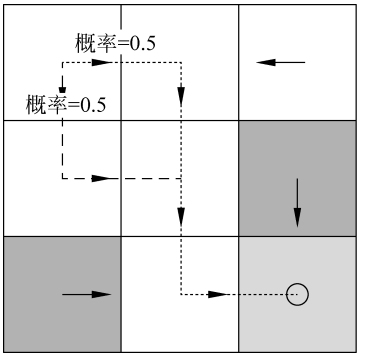

图 5-21　网格世界随机性策略示意图

图 5-21 所示的策略可以用 π 表示：

$$\begin{aligned}&\pi(a_1 \mid s_1)=0\\&\pi(a_2 \mid s_1)=0.5\\&\pi(a_3 \mid s_1)=0.5\\&\pi(a_4 \mid s_1)=0\\&\pi(a_5 \mid s_1)=0\end{aligned} \tag{5-140}$$

Reward(回报)是一个标量值，用 r 来表示，它是一个实数。回报有时称为奖励，有时称为惩罚。一个行动发出后环境给智能体一个积极的反馈，则得到一个正值，若环境给智能体一个消极的反馈，则会得到一个负值(这种正值负值的回报参数设定只是一种常见设计，回报的设计取决于算法工程师对应用场景的理解，从而对此进行强化学习建模的设计。此外还存在不以正负值来设计的回报参数)。

对于图 5-17 的网格世界，可以采取这样的回报参数设定，若是某行动使状态跳出边界，则回报值为－1；若是某行动使状态进入禁止区域，则回报值为－1；若是某行动使状态到达目标区域，则回报值为＋1，其余情况的回报为＋0，即

$$\begin{aligned}&r_{\text{跳出边界}}=-1\\&r_{\text{进入禁区}}=-1\\&r_{\text{到达目标}}=1\\&r=0\end{aligned} \tag{5-141}$$

同样，回报的表格表达形式读者也可以参照表 5-2 来填写及修改，列名和行名不变，里

面每个单元格填入某状态通过某行动会得到的回报。

回报也可以用条件概率来表达，如 $p(r=1|s_1,a_1)=1$ 表示从 s_1 通过 a_1 得到回报 1 的概率为 1，$p(r\neq1|s_1,a_1)=0$ 表示从 s_1 通过 a_1 得到回报不为 1 的概率为 0。

一条行动轨迹是一条状态行动回报链，如图 5-22 所示。

$$s_1 \xrightarrow[r=0]{a_2} s_2 \xrightarrow[r=0]{a_3} s_5 \xrightarrow[r=0]{a_3} s_8 \xrightarrow[r=1]{a_2} s_9 \tag{5-142}$$

一条行动轨迹的回报是把沿着这个轨迹得到的所有回报和，图 5-22 所示的这条行动轨迹的回报公式为

$$\text{return}=0+0+0+1=1 \tag{5-143}$$

再来看这样一条行动回报轨迹，如图 5-23 所示。

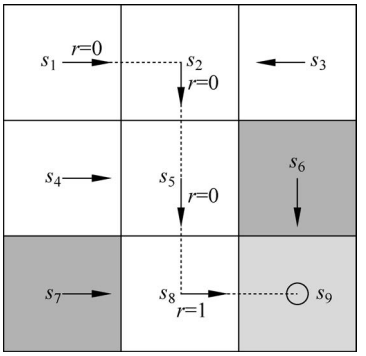

图 5-22　网格世界行动回报轨迹示意图(1)

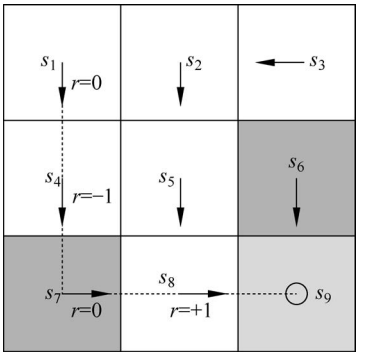

图 5-23　网格世界行动回报轨迹示意图(2)

这里有另一条行动轨迹，图 5-23 所示的这条行动轨迹

$$s_1 \xrightarrow[r=0]{a_3} s_4 \xrightarrow[r=-1]{a_3} s_5 \xrightarrow[r=0]{a_2} s_8 \xrightarrow[r=1]{a_2} s_9 \tag{5-144}$$

的回报公式为

$$\text{return}=0-1+0+1=0 \tag{5-145}$$

在图 5-22 和图 5-23 两条行动轨迹中，前者的回报大于后者的回报，可知图 5-22 是比图 5-23 更好的行动轨迹。这是由于后者进入过禁止区域，进入禁止区域的这个行动，在此设计的回报为－1。回报可以用于评估策略质量的好坏，但在比较回报的过程中，需要注意行动回报参数的设计是否合理，因为不合理的回报参数设计可能会得到荒谬的结论，如好的行动轨迹回报居然比差的行动轨迹的回报少。只有合理的回报参数设计才能使后续的策略搜索工作良性地持续下去。

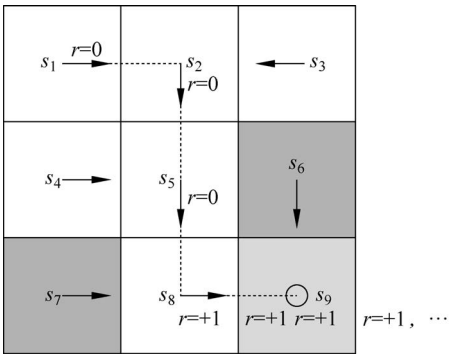

图 5-24　网格世界行动轨迹回报示意图

如图 5-24 所示的行动轨迹是一个无限轨迹，到达目标区域的时候，行动并没有停止，而是以待在原

地不动的行动持续地运行，这就使这条轨迹

$$s_1 \xrightarrow[r=0]{a_2} s_2 \xrightarrow[r=-1]{a_3} s_5 \xrightarrow[r=0]{a_3} s_8 \xrightarrow[r=1]{a_2} s_9 \xrightarrow[r=1]{a_5} s_9 \xrightarrow[r=1]{a_5} s_9 \cdots \tag{5-146}$$

的回报公式为

$$\text{return} = 0+0+0+1+1+1+\cdots = \infty \tag{5-147}$$

此回报沿着这个无穷长的轨迹发散下去，这就需要引入折扣因子 $\gamma \in [0,1)$。

折扣回报公式为

$$\begin{aligned}\text{discounted return} &= 0+\gamma 0+\gamma^2 0+\gamma^3 1+\gamma^4 1+\gamma^5 1+\cdots \\ &= \gamma^3(0+\gamma+\gamma^2+\gamma^3+\cdots) = \gamma^3\frac{1}{1-\gamma}\end{aligned} \tag{5-148}$$

可知 γ 越接近 0，未来回报衰减得越快，γ 越接近 1，未来回报衰减得越慢。通过控制 γ，可以控制智能体所学到的策略，γ 越小，就会越关注近期回报，反之，就会越关注长远回报。

相对于无限轨迹来讲，有限轨迹(回合制任务)指的是到达目标之后结束探索，有限探索任务，可以通过到达目标后持续地停留在目标区域，并把重复到达目标的回报修改为 0，从而转换为无限探索任务。这时目标状态可以被称为一种吸收状态，类似于马尔可夫链中的吸收态。更好的策略探索方式是使用折扣因子，无须将重复到达目标的回报修正为 0，使用折扣因子持续探索，可以使智能体在此之后发现这个策略不是最优策略的情况下从这个目标状态跳出来继续探索更优路径。这样相对来讲会耗费更多的搜索。

强化学习的任务和算法多种多样，这里介绍一些常见的分类，如图 5-25 所示。

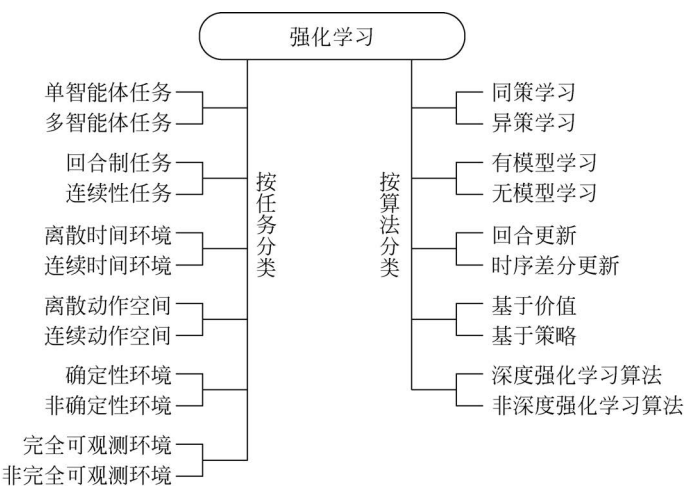

图 5-25　强化学习的常见分类

根据强化学习的任务和环境，可以对强化学习任务进行以下分类。

(1) 单智能体任务(Single Agent Task)和多智能体任务(Multi-Agent Task)：根据系统中的智能体数量，可以将任务划分为单智能体任务和多智能体任务。单智能体任务中只

有一个决策者，它可以得到所有可以观察到的观测，并能感知全局的奖励值；多智能体任务中有多个决策者，它们只能知道自己的观测，感受到环境给它的奖励。当然，在有需要的情况下，多个智能体间可以交换信息。在多智能体任务中，不同智能体奖励函数的不同会导致它们有不同的学习目标（甚至是互相对抗的）。

（2）回合制任务（Episodic Task）和连续性任务（Sequential Task）：对于回合制任务，可以有明确的开始状态和结束状态。例如在下围棋的时候，刚开始棋盘空空如也，最后棋盘都被摆满了，一局棋就可以看作一个回合。下一个回合开始时，一切重新开始。也有一些问题没有明确的开始和结束，例如机房的资源调度。机房从启用起就要不间断地处理各种信息，没有明确的结束又重新开始的时间点。

（3）离散时间环境（Discrete Time Environment）和连续时间环境（Continuous Time Environment）：如果智能体和环境的交互是分步进行的，就是离散时间环境。如果智能体和环境的交互是在连续的时间中进行的，就是连续时间环境。

（4）离散动作空间（Discrete Action Space）和连续动作空间（Continuous Action Space）：这是根据决策者可以做出的动作数量来划分的。如果决策得到的动作数量是有限的，则为离散动作空间，否则为连续动作空间。例如，走迷宫机器人如果只有东、南、西、北4种移动方向，则其为离散动作空间；如果机器人向360°中的任意角度都可以移动，则为连续动作空间。

（5）确定性环境任务（Deterministic Environment）和非确定性环境（Stochastic Environment）：按照环境是否具有随机性，可以将强化学习的环境分为确定性环境和非确定性环境。例如，对于机器人走固定的某个迷宫的问题，只要机器人确定了具体行动，那么结果就总是一成不变的。这样的环境就是确定性的，但是，如果迷宫会时刻随机变化，则机器人面对的环境就是非确定性的。

（6）完全可观测环境（Fully Observable Environment）和非完全可观测环境（Partially Observable Environment）：如果智能体可以观测到环境的全部知识，则环境是完全可观测的；如果智能体只能观测到环境的部分知识，则环境是非完全可观测的。例如，围棋问题就可以看作一个完全可观测的环境，因为可以看到棋盘的所有内容，并且假设对手总是用最优方法执行；扑克则不是完全可观测的，因为不知道对手手里有哪些牌。

从算法角度，可以对强化学习算法进行以下分类（这种分类方式对于追求强化学习数学原理的读者需要重点注意）。

（1）同策学习（On Policy）和异策学习（Off Policy）：同策学习是边决策边学习，学习者同时也是决策者。异策学习则是通过之前的历史（可以是自己的历史也可以是别人的历史）进行学习，学习者和决策者不需要相同。在异策学习的过程中，学习者并不一定要知道当时的决策。例如，围棋AI可以边对弈边学习，这就算同策学习；围棋AI也可以通过阅读人类的对弈历史来学习，这就是一个异策学习。

（2）有模型学习（Model-Based）和无模型学习（Model Free）：在学习的过程中，如果用到了环境的数学模型，则是有模型学习；如果没有用到环境的数学模型，则是无模型学习。

对于有模型学习，可能在学习前环境的模型就已经明确了，也可能环境的模型也是通过学习来获得的。例如，对于某个围棋 AI，它在下棋的时候可以在完全了解游戏规则的基础上虚拟出另外一个棋盘并在虚拟棋盘上试下，通过试下来学习。这就是有模型学习。与之相对，无模型学习不需要关于环境的信息，不需要搭建假的环境模型，所有经验都是通过与真实环境交互得到的。一个是可以在无数据的情况下基于模型学习，例如状态价值迭代算法、策略迭代算法；另一个则是可以在无模型的情况下基于数据学习，例如蒙特卡洛方法、时序差分方法。

(3) 回合更新(Monte Carlo Update)和时序差分更新(Temporal Difference Update)：回合更新是在回合结束后利用整个回合的信息进行更新学习，而时序差分更新不需要等回合结束，可以综合利用现有的信息和现有的估计进行更新学习。

(4) 基于价值(Value Based)和基于策略(Policy Based)：基于价值的强化学习定义了状态或动作的价值函数，以此来表示到达某种状态或执行某种动作后可以得到的回报。基于价值的强化学习倾向于选择价值最大的状态或动作；基于策略的强化学习算法不需要定义价值函数，它可以为行动分配概率分布，按照概率分布来执行行动。

(5) 深度强化学习(Deep Reinforcement Learning，DRL)算法和非深度强化学习算法。强化学习和深度学习是两种机器学习概念，两者可以结合。如果强化学习算法用到了深度学习，则这种强化学习可以称为深度强化学习算法。

至此，强化学习中的基本概念及如何分类已经一目了然，弄清楚状态、行动、回报和策略在强化学习中是至关重要的。

5.9.3 马尔可夫决策过程

前面部分通过一些例子说明了强化学习的一些基本概念。本节在马尔可夫决策过程(MDP)的框架下以更正式的方式介绍这些概念。MDP 是描述随机动力系统的通用框架。下面列出了 MDP 的关键要素。

1. 集合

状态空间：所有状态的集合，记为$\mathcal{S}$。

动作空间：所有行动的集合，表示为$\mathcal{A}(s)$，对于每种状态$s\in\mathcal{S}$。

奖励集：一组奖励，表示为$\mathcal{R}(s,a)$，对于每种状态行动对(s,a)。

2. 模型

状态转移概率：在状态s采取行动a的概率。转换到状态s'的概率为$p(s'|s,a)$。对于任意(s,a)，$\sum_{s'\in\mathcal{S}}p(r\mid s,a)=1$。

奖励概率：在状态s采取行动a时，被奖励的概率。奖励r为$p(r|s,a)$。对于任意(s,a)，$\sum_{r\in\mathcal{R}(s,a)}p(r\mid s,a)=1$。

策略：在状态s下，采取行动a的概率为$\pi(a|s)$，其中$\sum_{a\in\mathcal{A}(s)}\pi(a\mid s)=1\,\forall s\in\mathcal{S}$。

马尔可夫性质：马尔可夫性质是指随机过程的无记忆性质。从数学上来讲，这意味着：

$$p(s_{t+1} \mid s_t, \cdots, s_{t-1}, a_{t-1}, a_0, a_0) = p(s_{t+1} \mid s_t, a_t)$$
$$p(r_{t+1} \mid s_t, \cdots, s_{t-1}, a_{t-1}, a_0, a_0) = p(r_{t+1} \mid s_t, a_t) \tag{5-149}$$

其中，t 表示当前时间步长，$t+1$ 表示下一个时间步长。以上两个方程式表明下一种状态或奖励仅取决于当前状态和行动，与之前的状态和行动无关。马尔可夫性质对于推导 MDP 的基本贝尔曼方程(Bellman Equation)非常重要。

所有(s,a)的 $p(s'|s,a)$和 $p(r|s,a)$称为模型或动力系统。该模型可以是平稳的或非平稳的(或者换句话说，时不变的或时变的)。平稳模型不会随时间变化；非平稳模型可能会随时间的变化而变化。例如，在网格世界的例子中，如果禁区有时会弹出或消失，则模型是非平稳的。

马尔可夫决策过程(MDP)与马尔可夫过程(MP)不是一样的概念，一旦策略固定，马尔可夫决策过程就会退化为马尔可夫过程。

5.9.4　贝尔曼方程

贝尔曼方程由理查德·贝尔曼(Richard Bellman)发现。是关于未知函数(目标函数)的函数方程组。应用最优化原理和嵌入原理建立函数方程组的方法称为函数方程法。在实际运用中要按照具体问题寻求特殊解法。动态规划理论开拓了函数方程理论中许多新的领域。贝尔曼方程最早应用在工程领域的控制理论和其他应用数学领域，而后成为经济学上的重要工具。绝大多数可以用最佳控制理论(Optimal Control Theory)解决的问题也可以通过分析合适的贝尔曼方程得到解决，然而，贝尔曼方程通常指离散时间(Discrete-Time)最佳化问题的动态规划方程(Dynamic Programming Equation)。

在处理连续时间(Continuous-Time)最佳化问题上，也有一类重要的偏微分方程，称作 HJB 方程。

观察如图 5-26 所示的 3 个策略。

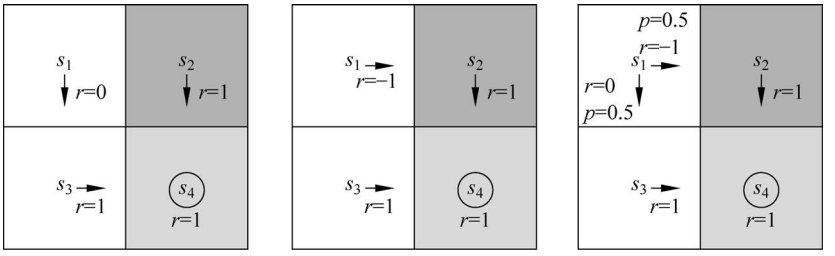

图 5-26　网格世界的 3 个策略

通过求解策略的奖励和并进行比较可以得出不同策略中相对最好的策略。

策略一的 return 值为

$$\begin{aligned} \text{return}_1 &= 0 + \gamma 1 + \gamma^2 1 + \cdots \\ &= \gamma(1 + \gamma + \gamma^2 + \cdots) \\ &= \frac{\gamma}{1-\gamma} \end{aligned} \tag{5-150}$$

策略二的 return 值为

$$\begin{aligned}\text{return}_2 &= -1+\gamma 1+\gamma^2 1+\cdots\\ &= -1+\gamma(1+\gamma+\gamma^2+\cdots)\\ &= -1+\frac{\gamma}{1-\gamma}\end{aligned} \tag{5-151}$$

策略三的 return 值为

$$\begin{aligned}\text{return}_3 &= 0.5\left(-1+\frac{\gamma}{1-\gamma}\right)+0.5\left(\frac{\gamma}{1-\gamma}\right)\\ &= 0.5+\frac{\gamma}{1-\gamma}\\ &= -1+\frac{\gamma}{1-\gamma}\end{aligned} \tag{5-152}$$

通过上述 3 个 return 可知对于任意的 γ,有

$$\text{return}_1 > \text{return}_2 > \text{return}_3 \tag{5-153}$$

表明第 1 个策略是最好的,因为它的奖励和是最大的,而第 2 个策略是最差的,因为它的回报是最小的。值得注意的是,return_3 并不严格遵守奖励的定义,它更像是一个期望值,如图 5-27 所示。

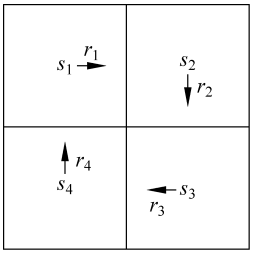

图 5-27 奖励公式轨迹图

奖励等于轨迹中所有奖励的折扣总和,另 v_i 表示 s_i 时获得的单次奖励,图 5-27 的轨迹例子中,奖励公式可以这样计算:

$$\begin{aligned}v_1 &= r_1+\gamma r_2+\gamma^2 r_3+\cdots\\ v_2 &= r_2+\gamma r_3+\gamma^2 r_4+\cdots\\ v_3 &= r_3+\gamma r_4+\gamma^2 r_1+\cdots\\ v_4 &= r_4+\gamma r_1+\gamma^2 r_2+\cdots\end{aligned} \tag{5-154}$$

通过简单的推导可以得出单次奖励公式为

$$\begin{aligned}v_1 &= r_1+\gamma(r_2+\gamma r_3+\cdots)=r_1+\gamma v_2\\ v_2 &= r_2+\gamma(r_3+\gamma r_4+\cdots)=r_2+\gamma v_3\\ v_3 &= r_3+\gamma(r_4+\gamma r_1+\cdots)=r_3+\gamma v_4\\ v_4 &= r_1+\gamma(r_1+\gamma r_2+\cdots)=r_4+\gamma v_1\end{aligned} \tag{5-155}$$

上述方程表明单次奖励在一条轨迹中相互依赖。方程可以改写为

$$\begin{bmatrix}v_1\\v_2\\v_3\\v_4\end{bmatrix}=\begin{bmatrix}r_1\\r_2\\r_3\\r_4\end{bmatrix}+\begin{bmatrix}\gamma v_2\\\gamma v_3\\\gamma v_4\\\gamma v_1\end{bmatrix}=\begin{bmatrix}r_1\\r_2\\r_3\\r_4\end{bmatrix}+\gamma\begin{bmatrix}0&1&0&0\\0&0&1&0\\0&0&0&1\\1&0&0&0\end{bmatrix}\begin{bmatrix}v_1\\v_2\\v_3\\v_4\end{bmatrix} \tag{5-156}$$

用矩阵形式可以写为

$$\boldsymbol{v}=\boldsymbol{r}+\gamma\boldsymbol{P}\boldsymbol{v} \tag{5-157}$$

因此，$\boldsymbol{v}$ 的值可以很容易地计算为 $\boldsymbol{v}=(\boldsymbol{I}-\gamma\boldsymbol{P})^{-1}\boldsymbol{r}$，其中 $\boldsymbol{I}$ 是单位矩阵。实际上这是贝尔曼公式的一个简单例子。$\boldsymbol{I}-\gamma\boldsymbol{P}$ 总是可逆的。对证明感兴趣的读者可以自己尝试证明或者查阅代数学相关书籍。

它们可以用来评估策略，但不适用于随机系统，因为从某种状态开始可能会导致不同的回报。

接下来引入状态值的概念。考虑时间序列 $t=1,2,3,\cdots$，在时间 t，智能体处于状态 S_t，并且遵循策略 π 采取的操作为 A_t。下一种状态是 S_{t+1}，即时反馈的奖励是 R_{t+1}。这个过程可以简明地表示为

$$S_t \xrightarrow{A_t} S_{t+1}, R_{t+1} \tag{5-158}$$

其中，S_t，S_{t+1}，A_t，A_{t+1} 都是随机变量，并且 $S_t, S_{t+1} \in \mathcal{S}$，$A_t \in \mathcal{A}(S_t)$，$R_{t+1} \in \mathcal{R}(S_t, A_t)$。

从 t 开始，有状态-动作-奖励轨迹：

$$S_t \xrightarrow{A_t} S_{t+1}, R_{t+1} \xrightarrow{A_{t+1}} S_{t+2}, R_{t+2} \xrightarrow{A_{t+2}} S_{t+3}, R_{t+3} \tag{5-159}$$

根据定义，沿轨迹的折扣奖励为

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots, \quad \gamma \in (0,1) \tag{5-160}$$

其中，G_t 是一个随机变量，因为 R_i 是随机变量，因此可以计算 G_t 的期望值（也称为期望或平均值）：

$$v_\pi(s) = E[G_t \mid S_t = s] \tag{5-161}$$

$v_\pi(s)$ 被称为状态值函数或者简称为 s 的状态值，其中 $v_\pi(s)$ 依赖 s，这是因为它的定义是一个条件期望，条件是智能体从 $S_t = s$ 开始。$v_\pi(s)$ 依赖 π，这是因为轨迹是通过遵循策略 π 生成的。对于不同的策略，状态值可能不同。$v_\pi(s)$ 不依赖于 t。如果智能体在状态空间中移动，则 t 表示当前时间步长。一旦给出策略，$v_\pi(s)$ 的值就确定了。

当策略和系统模型都是确定性的时，从一种状态开始总会导致相同的轨迹。在这种情况下，从一种状态开始获得的回报等于该状态的值。相比之下，当策略或系统模型是随机的时，从相同的状态开始可能会产生不同的轨迹。在这种情况下，不同轨迹的奖励是不同的，状态值是这些奖励的平均值。

奖励可以用来评估策略，然而使用状态值来评估策略更为正式：产生更大状态值（状态价值）的策略更好，因此，状态值构成了强化学习的核心概念。

贝尔曼方程是一种用于分析状态值的数学工具。简而言之，贝尔曼方程是一组描述所有状态值之间关系的线性方程。

折扣奖励公式可以改写为

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \\ &= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \cdots) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned} \tag{5-162}$$

其中，$G_{t+1} = R_{t+2} + \gamma R_{t+3} + \cdots$，该方程建立了 G_t 和 G_{t+1} 之间的关系。那么状态值可以

写为

$$\begin{aligned} v_\pi(s) &= E[G_t \mid S_t = s] \\ &= E[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\ &= E[R_{t+1} \mid S_t = s] + \gamma E[G_{t+1} \mid S_t = s] \end{aligned} \tag{5-163}$$

这样,状态值函数被写成了两项期望值之和。首先来看第 1 项期望值,它代表即时奖励期望,利用全期望定理,它可以写为

$$\begin{aligned} E[R_{t+1} \mid S_t = s] &= \sum_{a \in A} \pi(a \mid s) E[R_{t+1} \mid S_t = s, A_t = a] \\ &= \sum_{a \in A} \pi(a \mid s) \sum_{r \in R} p(r \mid s, a) r \end{aligned} \tag{5-164}$$

其中,A 和 R 分别是可能的动作和奖励的集合。需要注意的是,对于不同的状态,A 可能不同。在这种情况下,A 应写为 $A(s)$。类似地,R 也取决于(s,a)。尽管如此,在存在依赖性的情况下,结论仍然有效。

第 2 项期望值代表未来奖励期望,可以写为

$$\begin{aligned} E[G_{t+1} \mid S_t = s] &= \sum_{s' \in S} E[G_{t+1} \mid S_t = s, S_{t+1} = s'] p(s' \mid s) \\ &= \sum_{s' \in S} E[G_{t+1} \mid S_{t+1} = s'] p(s' \mid s) \\ &= \sum_{s' \in S} v_\pi(s') p(s' \mid s) \\ &= \sum_{s' \in S} v_\pi(s') \sum_{a \in A} p(s' \mid s, a) \pi(a \mid s). \end{aligned} \tag{5-165}$$

上述公式推导中的第 2 步使用了马尔可夫性质。这意味着未来奖励值只取决于当前状态,而不依赖于之前的状态。

有了以上两个期望值公式,可以把状态价值函数展开写为

$$\begin{aligned} v_\pi(s) &= E[R_{t+1} \mid S_t = s] + \gamma E[G_{t+1} \mid S_t = s], \\ &= \underbrace{\sum_{a \in A} \pi(a \mid s) \sum_{r \in R} p(r \mid s, a) r}_{\text{mean of immediate rewards}} + \underbrace{\gamma \sum_{a \in A} \pi(a \mid s) \sum_{s' \in S} p(s' \mid s, a) v_\pi(s')}_{\text{mean of future rewards}}, \\ &= \sum_{a \in A} \pi(a \mid s) \left[\sum_{r \in R} p(r \mid s, a) r + \gamma \sum_{s' \in S} p(s' \mid s, a) v_\pi(s') \right], \quad \text{for all } s \in S \end{aligned} \tag{5-166}$$

这个方程就是贝尔曼方程,它描述了状态值的关系。它是设计和分析强化学习算法的基本工具。

在这个贝尔曼方程中,$v_\pi(s)$和 $v_\pi(s')$是待计算的未知状态值。由于未知 $v_\pi(s)$依赖于另一个未知 $v_\pi(s')$,因此初学者可能会感到困惑,如何计算未知 $v_\pi(s)$。必须注意的是,贝尔曼方程是指所有状态的一组线性方程,而不是单个方程。

$\pi(a|s)$是给定的策略。由于状态值可以用来评估策略,因此从贝尔曼方程求解状态值

就是一个策略评估过程。

$p(r|s,a)$和 $p(s'|s,a)$代表系统模型。如何使用该模型计算状态值，这是一个在有模型的情况下计算状态值的场景(model-based)，还有一种场景是在无模型的情况下计算状态值(model-free)。

读者还可能在文献中遇到贝尔曼方程的其他表达式。接下来介绍两个等价的表达式。

首先有全概率定理：

$$p(s' \mid s,a)=\sum_{r\in R} p(s',r \mid s,a),$$
$$p(r \mid s,a)=\sum_{s'\in S} p(s',r \mid s,a) \tag{5-167}$$

可以把贝尔曼方程写为这种形式：

$$v_\pi(s)=\sum_{a\in A}\pi(a \mid s)\sum_{s'\in S}\sum_{r\in R} p(s',r \mid s,a)\left[r+\gamma v_\pi(s')\right] \tag{5-168}$$

其次，在某些问题中，奖励 r 可能仅取决于下一种状态 s'。作为一个结果，可以将奖励写为 $r(s')$，因此 $p(r(s')|s,a)=p(s'|s,a)$，这样，贝尔曼方程就可写为

$$v_\pi(s)=\sum_{a\in A}\pi(a \mid s)\sum_{s'\in S} p(s' \mid s,a)\left[r(s')+\gamma v_\pi(s')\right] \tag{5-169}$$

接下来用一个随机策略的例子来演示如何一步步地写出贝尔曼方程并计算状态值，如图 5-28 所示。

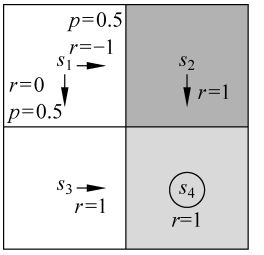

图 5-28　网格世界随机策略

首先需要写出贝尔曼方程，然后从中求解状态值。在状态 s_1 处，向右和向下的概率等于 0.5。有 $\pi(a=a_2,|s_1)=0.5$ 和 $\pi(a=a_3,|s_1)=0.5$。状态转移概率是确定性的，因为 $p(s'=s_3|s_1,a_3)=1$ 且 $p(s'=s_2|s_1,a_2)=1$。奖励概率也是确定性的，因为 $p(r=0|s_1,a_3)=1$ 且 $p(r=-1|s_1,a_2)=1$。将这些值代入贝尔曼方程式可以得出：

$$v_\pi(s_1)=0.5\left[0+\gamma v_\pi(s_3)\right]+0.5\left[-1+\gamma v_\pi(s_2)\right] \tag{5-170}$$

同理可得

$$v_\pi(s_2)=1+\gamma v_\pi(s_4),$$
$$v_\pi(s_3)=1+\gamma v_\pi(s_4)$$
$$v_\pi(s_4)=1+\gamma v_\pi(s_4) \tag{5-171}$$

状态值可以从上面的方程求解。由于方程很简单，手动求解状态值并得到

$$v_\pi(s_4)=\frac{1}{1-\gamma}$$
$$v_\pi(s_3)=\frac{1}{1-\gamma}$$
$$v_\pi(s_2)=\frac{1}{1-\gamma}$$
$$v_\pi(s_1)=0.5\left[0+\gamma v_\pi(s_3)\right]+0.5\left[-1+\gamma v_\pi(s_2)\right]=-0.5+\frac{\gamma}{1-\gamma} \tag{5-172}$$

此外，设置 $\gamma=0.9$ 可得

$$\begin{aligned}v_\pi(s_4)&=10\\v_\pi(s_3)&=10\\v_\pi(s_2)&=10\\v_\pi(s_1)&=-0.5+9=8.5\end{aligned}\tag{5-173}$$

由于它对每种状态都有效，因此可以将所有这些方程结合起来，并将它们简洁地写成矩阵向量形式，这将经常用于分析贝尔曼方程。

为了导出矩阵向量形式，首先将贝尔曼方程重写为

$$v_\pi(s)=r_\pi(s)+\gamma\sum_{s'\in S}p_\pi(s'\mid s)v_\pi(s')\tag{5-174}$$

其中，

$$\begin{aligned}r_\pi(s)&=\sum_{a\in A}\pi(a\mid s)\sum_{r\in R}p(r\mid s,a)r\\p_\pi(s'\mid s)&=\sum_{a\in A}\pi(a\mid s)p(s'\mid s,a)\end{aligned}\tag{5-175}$$

这里，$r_\pi(s)$表示即时奖励的均值，$p_\pi(s'|s)$是在策略π下从s转移到s'的概率。

假设状态索引为 s_i，其中 $i=1,\cdots,n$，其中 $n=|S|$。对于状态 s_i，贝尔曼方程可以写为

$$v_\pi(s_i)=r_\pi(s_i)+\gamma\sum_{s_j\in S}p_\pi(s_j\mid s_i)v_\pi(s_j)\tag{5-176}$$

设 $\boldsymbol{v}_\pi=[v_\pi(s_1),\cdots,v_\pi s_n]^{\mathrm{T}}\in\mathbf{R}^n$，$\boldsymbol{r}_\pi=[r_\pi(s_1),\cdots,r_\pi(s_n)]^{\mathrm{T}}\in\mathbf{R}^n$，$\boldsymbol{P}_\pi\in\mathbf{R}^{n\times n}$，$[\boldsymbol{P}_\pi]_{ij}=p_\pi(s_j|s_i)$，则可写出贝尔曼方程的向量形式：

$$\boldsymbol{v}_\pi=\boldsymbol{r}_\pi+\gamma\boldsymbol{P}_\pi\boldsymbol{v}_\pi\tag{5-177}$$

其中，$\boldsymbol{v}_\pi$ 是待解的未知数，$\boldsymbol{r}_\pi$，$\boldsymbol{P}_\pi$ 已知。$\boldsymbol{P}_\pi$ 是一个非负矩阵，意味着它的所有元素都大于或等于零。此属性表示为 $\boldsymbol{P}_\pi\geqslant 0$，其中 0 表示具有适当维度的零矩阵。≥或≤代表元素比较操作。它还是一个随机矩阵，意味着每行中的值之和等于 1。该属性表示为 $\boldsymbol{P}_\pi\mathbf{1}=\mathbf{1}$，其中 $\mathbf{1}=[1,\cdots,1]^{\mathrm{T}}$ 具有适当的维度。

在图 5-14 的示例中，贝尔曼方程的矩阵向量形式为

$$\underbrace{\begin{bmatrix}v_\pi(s_1)\\v_\pi(s_2)\\v_\pi(s_3)\\v_\pi(s_4)\end{bmatrix}}_{v_\pi}=\underbrace{\begin{bmatrix}r_\pi(s_1)\\r_\pi(s_2)\\r_\pi(s_3)\\r_\pi(s_4)\end{bmatrix}}_{r_\pi}+\gamma\underbrace{\begin{bmatrix}p_\pi(s_1\mid s_1)&p_\pi(s_2\mid s_1)&p_\pi(s_3\mid s_1)&p_\pi(s_4\mid s_1)\\p_\pi(s_1\mid s_2)&p_\pi(s_2\mid s_2)&p_\pi(s_3\mid s_2)&p_\pi(s_4\mid s_2)\\p_\pi(s_1\mid s_3)&p_\pi(s_2\mid s_3)&p_\pi(s_3\mid s_3)&p_\pi(s_4\mid s_3)\\p_\pi(s_1\mid s_4)&p_\pi(s_2\mid s_4)&p_\pi(s_3\mid s_4)&p_\pi(s_4\mid s_4)\end{bmatrix}}_{P_\pi}\underbrace{\begin{bmatrix}v_\pi(s_1)\\v_\pi(s_2)\\v_\pi(s_3)\\v_\pi(s_4)\end{bmatrix}}_{v_\pi}\tag{5-178}$$

将具体值代入上式可得

$$\begin{bmatrix}v_\pi(s_1)\\v_\pi(s_2)\\v_\pi(s_3)\\v_\pi(s_4)\end{bmatrix}=\begin{bmatrix}0.5(0)+0.5(-1)\\1\\1\\1\end{bmatrix}+\gamma\begin{bmatrix}0&0.5&0.5&0\\0&0&0&1\\0&0&0&1\\0&0&0&1\end{bmatrix}\begin{bmatrix}v_\pi(s_1)\\v_\pi(s_2)\\v_\pi(s_3)\\v_\pi(s_4)\end{bmatrix}\tag{5-179}$$

可见 $\boldsymbol{P}_\pi$ 满足 $\boldsymbol{P}_\pi \mathbf{1}=1$。

计算给定策略的状态值是强化学习中的一个基本问题，这个问题通常被称为策略评估。

由于 $\boldsymbol{v}_\pi=\boldsymbol{r}_\pi+\gamma \boldsymbol{P}_\pi \boldsymbol{v}_\pi$ 是一个简单的线性方程，因此可以轻松地获得其闭式解：

$$\boldsymbol{v}_\pi=(\boldsymbol{I}-\gamma \boldsymbol{P}_\pi)^{-1}\boldsymbol{r}_\pi \tag{5-180}$$

其中，$\boldsymbol{I}-\gamma \boldsymbol{P}_\pi$ 是可逆的，盖尔圆盘定理(Gershgorin Circle Theorem)可以证明这点，且 $(\boldsymbol{I}-\gamma \boldsymbol{P}_\pi)^{-1}\geqslant \boldsymbol{I}$，这意味着 $(\boldsymbol{I}-\gamma \boldsymbol{P}_\pi)^{-1}$ 的每个元素都是非负的。

虽然闭式解对于理论分析很有用，但在实际中并不适用，因为它涉及矩阵求逆运算，仍然需要通过其他数值算法来计算。事实上更方便程序计算的是使用以下迭代算法直接求解贝尔曼方程：

$$\boldsymbol{v}_{k+1}=\boldsymbol{r}_\pi+\gamma \boldsymbol{P}_\pi \boldsymbol{v}_k \tag{5-181}$$

该算法会生成一个值序列 $\{v_0,v_1,v_2,\cdots\}$，其中 $v_0\in\mathbf{R}^n$ 是 v_π 的初始猜测值，且

$$\lim_{k\to\infty}\boldsymbol{v}_k=\boldsymbol{v}_\pi=(\boldsymbol{I}-\gamma \boldsymbol{P}_\pi)^{-1}\boldsymbol{r}_\pi \tag{5-182}$$

目前为止一直在讨论状态值，是时候介绍动作值了，它表示在某种状态下采取的行动的价值。虽然行动价值的概念很重要，之所以在最后引入它，是因为它严重依赖于状态价值的概念。在研究行动价值之前，首先要充分理解状态价值，这一点很重要。

状态-行动对 (s,a) 的动作值定义为

$$q_\pi(s,a)=E[G_t \mid S_t=s,A_t=a] \tag{5-183}$$

可以看出，动作值定义为在某种状态下采取行动后可以获得的预期奖励。必须注意的是，$q_\pi(s,a)$ 取决于状态-行动对 (s,a)，而不是单独的行动。将此值称为状态动作值可能更严格，但为了简单起见，通常将其称为动作值。

行动价值和状态价值之间的关系可以由条件期望得出：

$$\underbrace{E[G_t \mid S_t=s]}_{v_\pi(s)}=\sum_{a\in A}\underbrace{E[G_t \mid S_t=s,A_t=a]}_{q_\pi(s,a)}\pi(a \mid s) \tag{5-184}$$

从而得出：

$$v_\pi(s)=\sum_{a\in A}\pi(a \mid s)q_\pi(s,a) \tag{5-185}$$

因此，状态值是与该状态相关联的动作值的期望。它展示了如何从动作值获取状态值。

由于状态值由下式给出：

$$v_\pi(s)=\sum_{a\in A}\pi(a \mid s)\left[\sum_{r\in R}p(r \mid s,a)r+\gamma\sum_{s'\in S}p(s' \mid s,a)v_\pi(s')\right] \tag{5-186}$$

所以：

$$q_\pi(s,a)=\sum_{r\in R}p(r \mid s,a)r+\gamma\sum_{s'\in S}p(s' \mid s,a)v_\pi(s') \tag{5-187}$$

可以看出，动作值由两项组成。第 1 项是当前奖励的均值，第 2 项是未来奖励的均值。它展示了如何从状态值获取动作值。

下面通过一个例子来说明计算动作值的过程，并讨论初学者可能犯的一个常见错误。

继续以图 5-16 为例，先只检查 s_1 的行动，其他状态也可以类似地进行检查。(s_1,a_2)的动作值为

$$q_\pi(s_1,a_2)=-1+\gamma v_\pi(s_2) \tag{5-188}$$

其中，s_2 是下一种状态。同理可得

$$q_\pi(s_1,a_3)=0+\gamma v_\pi(s_3) \tag{5-189}$$

初学者可能犯的一个常见错误是关于给定策略未选择的动作值。例如，图 5-14 中的策略只能选择 a_2 或 a_3，不能选择 a_1、a_4、a_5。有人可能会说，既然策略没有选择 a_1、a_4、a_5，就不需要计算它们的动作值，或者简单地设置 $q_\pi(s_1,a_1)=q_\pi(s_1,a_4)=q_\pi(s_1,a_5)=0$，这是错误的。

即使某个行动不会被策略选择，它仍然具有操作值。在这个例子中，虽然策略 π 在 s_1 处没有采取 a_1，但仍可以通过观察采取该行动后得到的结果来计算其动作值。具体来讲，在获取 a_1 之后，智能体被弹回到 s_1(因此，即时奖励为-1)，然后继续从 s_1 开始沿着 π 在状态空间中移动(因此，未来奖励为 $\gamma v_\pi(s_1)$)，因此，(s_1,a_1)的动作值为

$$q_\pi(s_1,a_1)=-1+\gamma v_\pi(s_1) \tag{5-190}$$

类似地，对于 a_4 和 a_5，给定的策略也不可能选择它们，有

$$\begin{aligned} q_\pi(s_1,a_4)&=-1+\gamma v_\pi(s_1)\\ q_\pi(s_1,a_5)&=0+\gamma v_\pi(s_1) \end{aligned} \tag{5-191}$$

为什么需要关心给定策略不会选择的行动？尽管某些动作不可能被给定的策略选择，但这并不意味着这些行动不好。给定的策略可能不好，因此无法选择最佳操作。强化学习的目的是寻找最优策略。为此，必须不断地探索所有行动，以确定每种状态有更好的行动。

计算出动作值后，还可以根据动作值计算状态值：

$$\begin{aligned} v_\pi(s_1)&=0.5q_\pi(s_1,a_2)+0.5q_\pi(s_1,a_3)\\ &=0.5[0+\gamma v_\pi(s_3)]+0.5[-1+\gamma v_\pi(s_2)] \end{aligned} \tag{5-192}$$

之前介绍的贝尔曼方程是根据状态值定义的，其实也可以用动作值表达。

$$q_\pi(s,a)=\sum_{r\in R}p(r\mid s,a)r+\gamma\sum_{s'\in S}p(s'\mid s,a)\sum_{a'\in A(s')}\pi(a'\mid s')q_\pi(s',a') \tag{5-193}$$

这是动作值的方程，上面的方程对于每种状态-动作对都有效。若将所有这些方程都放在一起，则它们的矩阵向量形式为

$$\boldsymbol{q}_\pi=\tilde{\boldsymbol{r}}+\gamma\boldsymbol{P}\boldsymbol{\Pi}\boldsymbol{q}_\pi \tag{5-194}$$

其中，$\boldsymbol{q}_\pi$ 是由状态-行动对索引的动作值向量：其第(s,a)个元素是 $[q_\pi]_{(s,a)}=q_\pi(s,a)$。$\tilde{r}$ 是由状态-行动对索引的即时奖励向量：$[\tilde{\boldsymbol{r}}]_{(s,a)}=\sum_{r\in R}p(r\mid s,a)r$。

矩阵 $\boldsymbol{P}$ 是概率转置矩阵，其行由状态-行动对索引，其列由状态索引：$[P]_{(s,a,s')}=p(s'|s,a)$。此外，$\boldsymbol{\Pi}$ 是块对角矩阵，其中每个块是一个 $1\times|\boldsymbol{A}|$ 向量：$\boldsymbol{\Pi}_{s',(s',a')}=\pi(a'|s')$且$\boldsymbol{\Pi}$ 的其他项为 0。

与根据状态值定义的贝尔曼方程相比，根据动作值定义的方程有一些独有的特征。例

如，$\tilde{\boldsymbol{r}}$ 和 $\boldsymbol{P}$ 与策略无关，仅由系统模型决定。该策略嵌入在 $\boldsymbol{\Pi}$ 中。可以验证 $\boldsymbol{q}_\pi = \tilde{r} + \gamma \boldsymbol{P}\boldsymbol{\Pi}\boldsymbol{q}_\pi$ 也是收缩映射(Contraction Mapping)，并且有唯一解，可以迭代求解。

状态值是智能体从某种状态开始可以获得的预期回报。不同状态的值是相互关联的。也就是说，状态 s 的值依赖于其他一些状态的值，而其他状态的值可能进一步依赖于状态 s 本身的值。这一点会令人困惑，使用贝尔曼方程的矩阵向量形式，就会很清楚这点。

在寻找最优策略时，动作值比状态价值发挥着更直接的作用。另外，贝尔曼方程并不局限于强化学习领域。相反，它广泛存在于控制理论和运筹学等许多领域。在不同的领域，贝尔曼方程可能有不同的表达方式。这里介绍的贝尔曼方程是离散马尔可夫决策过程下的贝尔曼方程。

5.9.5　贝尔曼最优方程

网格世界策略的改进如图 5-29 所示。

策略的贝尔曼方程为

$$\begin{aligned} v_\pi(s_1) &= -1 + \gamma v_\pi(s_2), \\ v_\pi(s_2) &= +1 + \gamma v_\pi(s_4), \\ v_\pi(s_3) &= +1 + \gamma v_\pi(s_4), \\ v_\pi(s_4) &= +1 + \gamma v_\pi(s_4). \end{aligned} \tag{5-195}$$

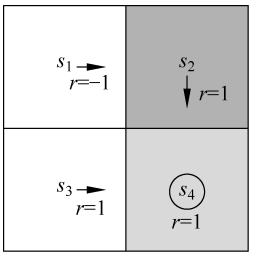

图 5-29　网格世界策略的改进

设折扣因子 $\gamma=0.9$，可以得出：

$$\begin{aligned} &v_\pi(s_4) = v_\pi(s_3) = v_\pi(s_2) = 10 \\ &v_\pi(s_1) = 8 \end{aligned} \tag{5-196}$$

对于状态 s_1 的动作值可以计算出：

$$\begin{aligned} q_\pi(s_1, a_1) &= -1 + \gamma v_\pi(s_1) = 6.2 \\ q_\pi(s_1, a_2) &= -1 + \gamma v_\pi(s_2) = 8 \\ q_\pi(s_1, a_3) &= 0 + \gamma v_\pi(s_3) = 9 \\ q_\pi(s_1, a_4) &= -1 + \gamma v_\pi(s_1) = 6.2 \\ q_\pi(s_1, a_5) &= 0 + \gamma v_\pi(s_1) = 7.2 \end{aligned} \tag{5-197}$$

注意，这里可以看到在状态 s_1 的所有行动中，a_3 这个行动具有最大的价值。

$$q_\pi(s_1, a_3) \geqslant q_\pi(s_1, a_i), \quad \text{对于所有 } i \tag{5-198}$$

因此可以更新策略以在 s_1 处选择 a_3。这个例子说明更新策略以选择具有最大动作值的行动，可以获得更好的策略。

最优策略的定义基于状态值。特别是，考虑两个给定的策略 π_1 和 π_2。如果任意状态下 π_1 的状态值大于或等于 π_2 的状态值：

$$v_{\pi_1}(s) \geqslant v_{\pi_2}(s), \quad \text{对于所有 } s \in S \tag{5-199}$$

那么 π_1 就比 π_2 好。另外，如果一项策略优于所有其他可能的策略，则这个策略就是最优的。

最优策略中的最优状态值的定义：如果对于所有 $s\in S$ 和任何其他策略 π，$v_{\pi^\times}(s)\geqslant v_\pi(s)$，则策略 $\pi^\times$ 是最优的。$\pi^\times$ 的状态值是最优状态值。

分析最优策略和最优状态值的工具是贝尔曼最优方程(Bellman Optimality Equation，BOE)。通过求解这个方程可以获得最优策略和最优状态值。

对于任意的 $s\in S$，有

$$\begin{aligned}v(s)&=\max_{\pi(s)\in\Pi(s)}\sum_{a\in A}\pi(a\mid s)\Big(\sum_{r\in R}p(r\mid s,a)r+\gamma\sum_{s'\in S}p(s'\mid s,a)v(s')\Big)\\&=\max_{\pi(s)\in\Pi(s)}\sum_{a\in A}\pi(a\mid s)q(s,a)\end{aligned}\tag{5-200}$$

其中，$v(s)$，$v(s')$ 是待求解的未知变量，此外

$$q(s,a)=\sum_{r\in R}p(r\mid s,a)r+\gamma\sum_{s'\in S}p(s'\mid s,a)v(s')\tag{5-201}$$

这里，$\pi(s)$ 表示状态 s 的策略，$\pi(s)$ 是 s 的所有可能策略的集合。

可以把贝尔曼最优方程简写为

$$v(s)=\max_{\pi(s)\in\Pi(s)}\sum_{a\in A}\pi(a\mid s)q(s,a),\quad s\in S\tag{5-202}$$

由于 $\sum_a\pi(a\mid s)=1$，所以有

$$\sum_{a\in A}\pi(a\mid s)q(s,a)\leqslant\sum_{a\in A}\pi(a\mid s)\max_{a\in A}q(s,a)=\max_{a\in A}q(s,a)\tag{5-203}$$

其中等号满足的条件是：

$$\pi(a\mid s)=\begin{cases}1, & a=a^\times\\ 0, & a\neq a^\times\end{cases}\tag{5-204}$$

这里 $a^\times=\arg\max_a q(s,a)$，总而言之，最优策略 $\pi(s)$ 是选择具有最大 $q(s,a)$ 值的动作的策略。

贝尔曼最优方程的向量形式：

$$\boldsymbol{v}=\max_{\pi\in\Pi}(\boldsymbol{r}_\pi+\gamma\boldsymbol{P}_\pi\boldsymbol{v})\tag{5-205}$$

根据收缩映射定理，可以将其表示为

$$f(\boldsymbol{v})=\max_{\pi\in\Pi}(\boldsymbol{r}_\pi+\gamma\boldsymbol{P}_\pi\boldsymbol{v})\tag{5-206}$$

可以看出，根据收缩映射定理，可推广 $\boldsymbol{v}$ 至 $f(\boldsymbol{v})$。

5.10 模型介绍

在本案例中，使用的是 A2C(Advantage Actor-Critic)算法，这是一个经典的无模型学习(Model-Free Learning)方法。A2C 模型是一种用于强化学习的机器学习模型，它结合了两种重要的概念：演员(Actor)和评论家(Critic)。

演员就像是模型的行为决策者。它负责在每个时间步骤上选择动作，就像一个演员在

戏剧中选择角色扮演一样。演员根据当前的状态和策略来选择动作，策略是一个函数，它告诉演员在给定状态下选择哪个动作。

评论家是模型的价值估计者。它的任务是评估每种状态的好坏程度，就像戏剧评论家评估戏剧的质量一样。评论家根据当前状态估计该状态的值或价值。价值表示在该状态下能够获得多大的回报或奖励。

A2C 模型的目标是通过不断地与环境互动来学习一个最佳的策略，以使累积回报最大化。这是通过以下方式实现的。

(1) 动作选择：演员使用当前状态和策略来选择一个动作。策略是一个概率分布，它告诉演员选择每个动作的概率。

(2) 环境互动：演员执行所选的动作，并与环境互动。环境会根据动作提供反馈信号，包括奖励信号。

(3) 奖励信号：根据与环境的互动，模型会获得奖励信号。奖励信号告诉模型每个动作的好坏程度。

(4) 更新策略：根据获得的奖励信号，模型会更新策略，以便在类似的情境中选择更好的动作。这是通过最大化奖励信号实现的。

(5) 值函数估计：评论家使用价值函数来估计每种状态的价值。这有助于模型更好地理解环境，并指导演员选择更好的动作。

总体来讲，A2C 是一种结合了演员和评论家的强化学习方法，用于训练一个智能代理来在不断的互动中学习如何做出最佳决策。这个模型已经在许多领域取得了成功，包括游戏、金融交易和机器人控制等。

5.10.1　数据准备

首先准备好市场的成交数据和订单簿数据，最好使用 websocket 网络协议接口获取。例如这里，获取了某品种的订单簿数据：orderbook.csv，如图 5-30 所示。

receive_ts	exchange_ts	bid_price_0	bid_vol_0	...	ask_price_9	ask_vol_9
2023/09/12 08:50:40.992	2023/09/12 08:50:40.976	float	float	...	float	float
2023/09/12 08:50:41.094	2023/09/12 08:50:40.079	float	float	...	float	float
2023/09/12 08:50:41.194	2023/09/12 08:50:40.171	float	float	...	float	float
2023/09/12 08:50:41.294	2023/09/12 08:50:40.282	float	float	...	float	float
...	...	...	...	...	...	...
2023/09/14 01:23:48.580	2023/09/14 01:23:48.569	float	float	...	float	float

图 5-30　订单簿数据

及它的成交数据：trade.csv，如图 5-31 所示。

receive_ts	exchange_ts	aggro_side	price	size
2023/09/12 08:50:41.763	2023/09/12 08:50:41.747	ASK	float	float
2023/09/12 08:50:43.877	2023/09/12 08:50:44.000	ASK	float	float
2023/09/12 08:50:46.716	2023/09/12 08:50:47.000	ASK	float	float
...	...	...	...	...
2023/09/14 01:23:27.723	2023/09/14 01:23:27.704	BID	float	float

图 5-31　交易数据

运行环境还需要安装一些模块,最好让这一切运行在一个独立的运行环境中,推荐Docker或Conda。本例是用PyTorch来构建神经网络模型的,读者不妨尝试一下TensorFlow,代码如下:

```
#//第5章/get_features.ipynb
#使用mamba安装PyTorch
#macOS版
mamba install pytorch::pytorch torchvision torchaudio -c pytorch
#Linux版CPU
mamba install pytorch torchvision torchaudio cpuonly -c pytorch
#Linux版GPU
mamba install pytorch torchvision torchaudio pytorch-cuda=11.8 -c pytorch
-c nvidia
#Windows版CPU
mamba install pytorch torchvision torchaudio cpuonly -c pytorch
#Windows版GPU
mamba install pytorch torchvision torchaudio pytorch-cuda=11.8 -c pytorch
-c nvidia
#接下来需要一张用于生成环境的状态空间的features表
#首先需要读入订单簿、成交数据
import pandas as pd
lobs = './data/orderbook.csv'
trades = './data/trades.csv'
lobs_df = pd.read_csv(lobs)
trades_df = pd.read_csv(trades)
#修改时间戳格式
lobs_df['receive_ts'] = pd.to_datetime(lobs_df['receive_ts'], format='ISO8601')
lobs_df['exchange_ts'] = pd.to_datetime(lobs_df['exchange_ts'], format=
'ISO8601')
trades_df['receive_ts'] = pd.to_datetime(trades_df['receive_ts'], format=
'ISO8601')
trades_df['exchange_ts'] = pd.to_datetime(trades_df['exchange_ts'],
```

5.10.2 特征工程

以receive_ts字段生成features表,代码如下:

```
features_df = pd.DataFrame({'receive_ts': lobs_df['receive_ts']})
```

从订单簿数据中计算出中间价,代码如下:

```
lobs_df['mid_price'] = (lobs_df['ask_price_0'] + lobs_df['bid_price_0']) / 2
```

从成交数据中分离出三类数据作为一个列表,代码如下:

```
trades_df['trade'] = trades_df[['aggro_side', 'price', 'size']].values.tolist()
```

计算每一档价格到中间价的距离并放入features表中,代码如下:

```
for column in lobs_df.filter(regex="_price_").columns.values:
    features_df[f'dist_{column}'] = (lobs_df[column] / lobs_df['mid_price'] - 1) *
1e06
```

现在在 features 表中，除了时间戳字段 receive_ts 之外，就有 20 个字段，分别如下：dist_bid_price_0，dist_ask_price_0，dist_bid_price_1，dist_ask_price_1，……，dist_bid_price_9，dist_ask_price_9。

$$\text{dist\_x\_price}_i = \frac{\text{x\_price}_i}{\text{midprice}} - 1, \quad x \in \{\text{bid},\text{ask}\}, i \in [0,9] \cap \mathbf{Z} \tag{5-207}$$

计算不同挡位的累计名义价值并加入 features 表中：

```
#//第 5 章/get_features.ipynb
for side in ['bid', 'ask']:
    features_df[f'{side}_cumul_0'] = lobs_df[f'{side}_price_0'] * lobs_df[f'
{side}_vol_0']

for i in range(1, 10):
    for side in ['bid', 'ask']:
        features_df[f'{side}_cumul_{i}'] = (
          features_df[f'{side}_cumul_{i - 1}'] + lobs_df[f'{side}_price_{i}'] *
lobs_df[f'{side}_vol_{i}'])
```

$$\begin{aligned} &\text{x\_cumul}_0 = p_0^x v_0^x \\ &\text{x\_cumul}_i = \text{x\_cumul}_{i-1} + p_i^x v_i^x \\ &x \in \{\text{bid},\text{ask}\}, i \in [1,9] \cap \mathbf{Z} \end{aligned} \tag{5-208}$$

现在 features 表又增加了 20 个字段，分别是 bid_cumul_0，ask_cumul_0，…，ask_cumul_9。

计算市场不平衡度指标（Market Imbalance Indicator）或者简称不平衡度（Imbalance），这个比率通常用来度量买方和卖方之间的市场深度或订单簿中的不平衡，代码如下：

```
for i in range(10):
    features_df[f'national_imbalance_{i}'] = (
    (features_df[f'ask_cumul_{i}'] - features_df[f'bid_cumul_{i}']) / (features_df
[f'ask_cumul_{i}'] + features_df[f'bid_cumul_{i}']))
```

现在 features 表又多了 10 个字段，分别是 national_imbalance_0，national_imbalance_1，…，national_imbalance_9。

接下来计算订单流平衡度指标并加入 features 表中，代码如下：

```
#//第 5 章/get_features.ipynb
bid_price = lobs_df['bid_price_0']
ask_price = lobs_df['ask_price_0']
bid_vol = lobs_df['bid_vol_0']
ask_vol = lobs_df['ask_vol_0']

prev_bid_price = lobs_df['bid_price_0'].shift(1)
prev_ask_price = lobs_df['ask_price_0'].shift(1)
prev_bid_vol = lobs_df['bid_vol_0'].shift(1)
prev_ask_vol = lobs_df['ask_vol_0'].shift(1)

features_df['order_flow_imbalance'] = (
  (bid_price >= prev_bid_price) *bid_vol -
```

```
    (bid_price <= prev_bid_price) *prev_bid_vol -
    (ask_price <= prev_ask_price) *ask_vol +
    (ask_price >= prev_ask_price) *prev_ask_vol)

features_df['order_flow_imbalance'].fillna(0, inplace=True)
```

订单流不平衡是金融市场中常用的一个指标,用于衡量买方和卖方之间的交易活动不平衡程度。该指标考虑了以下因素:

(1) 当买方报价(bid_price)大于或等于前一时刻的买方报价(prev_bid_price)时,增加了当前买方报价的数量(bid_vol)。

(2) 当买方报价小于或等于前一时刻的买方报价时,减少了前一时刻买方报价的数量(prev_bid_vol)。

(3) 当卖方报价(ask_price)小于或等于前一时刻的卖方报价(prev_ask_price)时,减少了当前卖方报价的数量(ask_vol)。

(4) 当卖方报价大于或等于前一时刻的卖方报价时,增加了前一时刻卖方报价的数量(prev_ask_vol)。

最终,通过以上计算,order_flow_imbalance 字段中的值表示在当前价格水平上的订单流不平衡程度。正值表示买方力量较强,负值表示卖方力量较强,0 表示相对平衡。这种指标通常用于分析市场中的买卖压力,以帮助预测价格走势或市场趋势。

$$\text{rise}^{b}bv_0 - \text{fall}^{b}bv_0^{-} - \text{rise}^{a}av_0 + \text{fall}^{a}av_0^{-} \tag{5-209}$$

其中,rise^{b}、fall^{b}、rise^{a}、fall^{a},代表相对市场环境的布尔值,分别代表买方市场上升、买方市场下降、卖方市场上升、卖方市场下降。如果成立,则为 1,如果不成立,则为 0。它们分别为买一价、先前买一价、卖一价、先前卖一价的无符号系数。

现在 features 表又多了一个字段 order_flow_imbalance。

再给 features 表加一个价差特征,代码如下:

```
features_df['spread'] = (lobs_df['ask_price_0'] - lobs_df['bid_price_0']) / lobs
_df['mid_price'] *1e06
```

$$\text{spread} = \frac{ap_0 - bp_0}{\text{midprice}} \tag{5-210}$$

现在 features 表又多了一个字段 spread。

接下来计算 RSI 指标,RSI 是一种用于衡量资产价格走势的技术指标,它通常用来识别市场是否处于超买(Overbought)或超卖(Oversold)的状态。RSI 的计算基于一定时期内的价格变化幅度,通常在 0 到 100 取值。我们以 1min,5min,15min 作为周期来计算,代码如下:

```
#//第 5 章/get_features.ipynb
lobs_df['returns'] = (lobs_df['mid_price'] / lobs_df['mid_price'].shift(1) - 1).
fillna(0)

windows = {'60s': '1m', '300s': '5m', '900s': '15m'}
```

```
    for w in tqdm(windows.keys()):
        lobs_df.loc[:, f'gain_{windows[w]}'] = lobs_df.set_index('receive_ts')
    ['returns'].rolling(w).apply(
        lambda x: x[x > 0].sum()
        ).reset_index()['returns']

        lobs_df.loc[:, f'loss_{windows[w]}'] = lobs_df.set_index('receive_ts')
    ['returns'].rolling(w).apply(
        lambda x: x[x < 0].sum()
        ).reset_index()['returns']
```

现在 features 表增加了 returns,gain_1m,loss_1m,gain_5m,loss_5m,gain_15m,loss_15m 字段。

接下来计算 CSRI(累计相对强弱指数),代码如下:

```
    #//第 5 章/get_features.ipynb
    for w in windows.values():
        features_df[f'CRSI_{w}'] = (
        (lobs_df[f'gain_{w}'] - lobs_df[f'loss_{w}'].abs()) /
        (lobs_df[f'gain_{w}'] + lobs_df[f'loss_{w}'].abs())
        ).fillna(0)
```

$$\mathrm{CRSI}=\frac{|\ \mathrm{gain}-\mathrm{loss}\ |}{|\ \mathrm{gain}+\mathrm{loss}\ |} \tag{5-211}$$

不同时间窗口的 CRSI 值可以提供有关不同时间尺度上市场趋势的信息,现在 features 表增加了 CRSI_1m,CRSI_5m,CRSI_15m 字段。在本例中,特征工程到此为止除去 receive_ts 字段一共有 55 个特征,为了方便后续的训练,可以把它存为一个特征表文件,代码如下:

```
    features_df['ESS'] = features_df.iloc[:, 1:].values.tolist()

    features_df.info()
```

查看数据结构,代码如下:

```
    #//第 5 章/get_features.ipynb
    Data columns (total 57 columns):
    #Column Non-Null Count Dtype
    --- ------ -------------- -----
    0 receive_ts 880834 non-null datetime64[ns]
    1 dist_bid_price_0 880834 non-null float64
    2 dist_ask_price_0 880834 non-null float64
    3 dist_bid_price_1 880834 non-null float64
    4 dist_ask_price_1 880834 non-null float64
    5 dist_bid_price_2 880834 non-null float64
    6 dist_ask_price_2 880834 non-null float64
    7 dist_bid_price_3 880834 non-null float64
    8 dist_ask_price_3 880834 non-null float64
    9 dist_bid_price_4 880834 non-null float64
    10 dist_ask_price_4 880834 non-null float64
    11 dist_bid_price_5 880834 non-null float64
    12 dist_ask_price_5 880834 non-null float64
    13 dist_bid_price_6 880834 non-null float64
```

```
14 dist_ask_price_6 880834 non-null float64
15 dist_bid_price_7 880834 non-null float64
16 dist_ask_price_7 880834 non-null float64
17 dist_bid_price_8 880834 non-null float64
18 dist_ask_price_8 880834 non-null float64
19 dist_bid_price_9 880834 non-null float64

...

55 CRSI_15m 880834 non-null float64
56 ESS 880834 non-null object
dtypes: datetime64[ns](1), float64(55), object(1)
```

把它写入 pickle 文件里,代码如下:

```
import pickle
features_dict = features_df.set_index('receive_ts')['ESS'].to_dict()
with open('./data/features_dict.pickle', 'wb') as f:
  pickle.dump(features_dict, f)
```

在训练的脚本中,可以通过以下代码载入特征数据,代码如下:

```
#//第 5 章/get_features.ipynb
with open('./data/features_dict.pickle', 'rb') as f:
    ess_dict = pickle.load(f)

#ess 代表环境状态空间 Environment State Space
ess_df = pd.DataFrame.from_dict(ess_dict,
               orient='index').reset_index().rename(columns
               {'index': 'receive_ts'})
```

现在还需要特征均值标准差的数据。后续在策略迭代过程中,还会给特征表增加动态数据库存比率及总盈亏。这两个特征是在 Agent 与 Environment 交互时实时计算的。对于这两个特征,将均值设置为 0,将标准差设置为 1,以保持一致性并且便于模型训练。为了方便,现在就先把均值标准差数据做出来,并存为 npy 文件,代码如下:

```
#//第 5 章/get_features.ipynb
import numpy as np

means = ess_df.mean()
stds = ess_df.std()

means = means.drop(columns='receive_ts').values
stds = stds.drop(columns='receive_ts').values

means = np.append(means, 0.0)
means = np.append(means, 0.0)
stds = np.append(stds, 1.0)
stds = np.append(stds, 1.0)

np.save('./data/means.npy', means)
np.save('./data/stds.npy', stds)
```

后续可以这样载入 npy 文件，代码如下：

```
#//第 5 章/get_features.ipynb
with open('./data/means.npy', 'rb') as f:
    means = np.load(f, allow_pickle=True)

with open('./data/stds.npy', 'rb') as f:
    stds = np.load(f, allow_pickle=True)
```

5.10.3　模型准备

首先介绍 simulator 模块，它用来模拟市场，我们就从这里切入主题。在训练模型中为了减少计算资源消耗，在 simulator 中定义了这样的数据类型，代码如下：

```
#//第 5 章//simulator/simulator.ipynb

from collections import deque
#导入 deque 集合，用于创建双端队列
from dataclasses import dataclass
#导入 dataclass，用于定义数据类
from typing import List, Optional, Tuple, Union, Deque, Dict
#导入类型提示相关的模块
import numpy as np
#导入 NumPy 库，用于处理数值数据
from sortedcontainers import SortedDict
#导入 SortedDict，用于创建有序字典
from datetime import datetime, timedelta
#导入 datetime 模块，用于处理日期和时间

#定义数据类以表示不同类型的交易和市场数据
@dataclass
class Order:                    #自己下的订单
    place_ts: float             #下单时间戳
    exchange_ts: float          #交易所(模拟器)接收订单的时间戳
    order_id: int
    side: str                   #买入('BID')或卖出('ASK')
    size: float
    price: float

@dataclass
class CancelOrder:
    exchange_ts: float
    id_to_delete: int           #要取消的订单的 ID

@dataclass
class AnonTrade:                #市场交易
    exchange_ts: float
    receive_ts: float
    side: str                   #买入('BID')或卖出('ASK')
    size: float
    price: float
```

```
@dataclass
class OwnTrade:                #执行自己下的订单
    place_ts: float            #调用 place_order 方法时的时间戳,用于调试
    exchange_ts: float
    receive_ts: float
    trade_id: int
    order_id: int
    side: str                  #买入('BID')或卖出('ASK')
    size: float
    price: float
    execute: str               #BOOK 或 TRADE,表示订单是通过撮合还是市场数据更新执行的

    def __post_init__(self):
        assert isinstance(self.side, str)

@dataclass
class OrderbookSnapshotUpdate:                  #订单簿快照更新
    exchange_ts: float
    receive_ts: float
    asks: List[Tuple[float, float]]             #列表,包含卖出订单的价格和数量元组
    bids: List[Tuple[float, float]]             #列表,包含买入订单的价格和数量元组

@dataclass
class MdUpdate:                                 #一个时间点的数据
    exchange_ts: float
    receive_ts: float
    orderbook: Optional[OrderbookSnapshotUpdate] = None    #可选的订单簿快照更新
    trade: Optional[AnonTrade] = None           #可选的市场交易数据
```

simulator 中还有一个全局函数,用于更新必要的最佳买卖位置及市场深度信息,代码如下:

```
#//第 5 章//simulator/simulator.ipynb

def update_best_positions(best_bid, best_ask, md: MdUpdate, levels: bool =
False) -> Tuple[float, float]:
    '''
    更新最佳买入和卖出位置及市场深度信息

    Args:
        best_bid (float): 当前最佳买入价格
        best_ask (float): 当前最佳卖出价格
        md (MdUpdate): 市场数据更新
        levels (bool): 是否返回市场深度信息

    Returns:
        Tuple[float, float]: 更新后的最佳买入和卖出价格
    '''
    if md.orderbook is not None:
        best_bid = md.orderbook.bids[0][0]                    #更新最佳买入价格
        best_ask = md.orderbook.asks[0][0]                    #更新最佳卖出价格
        if levels:
            asks = [level[0] for level in md.orderbook.asks] #卖出订单价格列表
            bids = [level[0] for level in md.orderbook.bids] #买入订单价格列表
```

```
            return best_bid, best_ask, asks, bids #返回最佳买入、卖出价格及市场深度
                                                  #信息
        else:
            return best_bid, best_ask #返回更新后的最佳买入和卖出价格
    else:
        if md.trade.side == 'BID':
            best_ask = max(md.trade.price, best_ask) #如果有市场交易,则根据交易更
                                                     #新最佳卖出价格
        elif md.trade.side == 'ASK':
            best_bid = min(best_bid, md.trade.price) #如果有市场交易,则根据交易更
                                                     #新最佳买入价格
        return best_bid, best_ask #返回更新后的最佳买入和卖出价格
```

Sim 类中的构造函数,代码如下:

```
#//第 5 章//simulator/simulator.ipynb

class Sim:
    def __init__(self, market_data: List[MdUpdate], execution_latency: float, md_
latency: float) -> None:
        '''
        初始化模拟交易系统

        Args:
            market_data (List[MdUpdate]): 市场数据列表,包含 MdUpdate 对象的列表
            execution_latency (float): 执行延迟,以纳秒为单位
            md_latency (float): 市场数据延迟,以纳秒为单位
        '''
        #将市场数据转换为队列
        self.md_queue = deque(market_data)
        #行动队列,用于存储订单和取消订单
        self.actions_queue: Deque[Union[Order, CancelOrder]] = deque()
        #SortedDict:receive_ts -> [updates],用于存储策略更新
        self.strategy_updates_queue = SortedDict()
        #映射:order_id -> Order,用于存储准备执行的订单
        self.ready_to_execute_orders: Dict[int, Order] = {}

        #当前市场数据
        self.md: Optional[MdUpdate] = None
        #当前订单 ID 和交易 ID
        self.order_id = 0
        self.trade_id = 0
        #延迟参数,包括执行延迟和市场数据延迟
        self.latency = execution_latency
        self.md_latency = md_latency
        #当前的最佳买入和卖出价格
        self.best_bid = -np.inf
        self.best_ask = np.inf
        #当前市场交易价格,包括最佳买入和卖出价格
        self.trade_price = {}
        self.trade_price['BID'] = -np.inf
        self.trade_price['ASK'] = np.inf
        #上一个订单,用于尝试主动执行
        self.last_order: Optional[Order] = None
```

这段代码创建了一个 Sim 类的实例,并初始化了该类的各个属性。这个类用于模拟交易系统,其中包括存储市场数据、订单队列、策略更新队列等信息。接下来的代码提供了一些方法来处理订单执行和市场数据更新等操作,代码如下:

```
#//第 5 章//simulator/simulator.ipynb
class Sim:
  def __init__(…):
    …

  def get_md_queue_event_time(self) -> np.float64:
      '''
      获取市场数据队列中下一个事件的时间戳(exchange_ts)

      Returns:
          np.float64: 下一个事件的时间戳,如果队列为空,则返回无穷大(np.inf)
      '''
      return np.inf if len(self.md_queue) == 0 else self.md_queue[0].exchange_ts


  def get_actions_queue_event_time(self) -> np.float64:
      '''
      获取行动队列中下一个事件的时间戳(exchange_ts)

      Returns:
          np.float64: 下一个事件的时间戳,如果队列为空,则返回无穷大(np.inf)
      '''
      return np.inf if len(self.actions_queue) == 0 else self.actions_queue[0].
exchange_ts


  def get_strategy_updates_queue_event_time(self) -> np.float64:
      '''
      获取策略更新队列中下一个事件的时间戳(receive_ts)

      Returns:
          np.float64: 下一个事件的时间戳,如果队列为空,则返回无穷大(np.inf)
      '''
       return np.inf if len(self.strategy_updates_queue) == 0 else list(self.
strategy_updates_queue.keys())[0]


  def get_order_id(self) -> int:
      '''
      获取唯一的订单 ID 并递增

      Returns:
          int: 唯一的订单 ID
      '''
      res = self.order_id
      self.order_id += 1
      return res
```

```
    def get_trade_id(self) -> int:
        '''
        获取唯一的交易 ID 并递增

        Returns:
            int: 唯一的交易 ID
        '''
        res = self.trade_id
        self.trade_id += 1
        return res


    def update_best_pos(self) -> None:
        '''
        更新最佳买入和卖出价格(best_bid 和 best_ask)

        Raises:
            AssertionError: 如果没有当前市场数据(self.md 为 None)
        '''
        assert not self.md is None, "no current market data!"
        if not self.md.orderbook is None:
            self.best_bid = self.md.orderbook.bids[0][0]
            self.best_ask = self.md.orderbook.asks[0][0]


    def update_last_trade(self) -> None:
        '''
        更新最近一笔交易的价格信息(trade_price)

        Raises:
            AssertionError: 如果没有当前市场数据(self.md 为 None)
        '''
        assert not self.md is None, "没有当前市场数据!"
        if not self.md.trade is None:
            self.trade_price[self.md.trade.side] = self.md.trade.price


    def delete_last_trade(self) -> None:
        '''
        删除最近一笔交易的价格信息(trade_price),将其重置为负无穷和正无穷
        '''
        self.trade_price['BID'] = -np.inf
        self.trade_price['ASK'] = np.inf


    def update_md(self, md: MdUpdate) -> None:
        '''
        更新当前市场数据(self.md)和策略更新队列(strategy_updates_queue)

        Args:
            md (MdUpdate): 新的市场数据

        Raises:
```

```
            AssertionError: 如果 md 中没有订单簿信息或交易信息
        '''
        #当前订单簿数据
        self.md = md
        #更新最佳买入和卖出价格信息
        self.update_best_pos()
        #更新最近一笔交易的价格信息
        self.update_last_trade()

        #将 md 添加到策略更新队列
        if not md.receive_ts in self.strategy_updates_queue.keys():
            self.strategy_updates_queue[md.receive_ts] = []
        self.strategy_updates_queue[md.receive_ts].append(md)


    def update_action(self, action: Union[Order, CancelOrder]) -> None:
        '''
        更新行动队列,包括存储订单和取消订单

        Args:
            action (Union[Order, CancelOrder]): 行动,可以是订单或取消订单

        Raises:
            AssertionError: 如果行动类型不正确
        '''
        if isinstance(action, Order):
            #存储最后一个订单,以便主动执行
            self.last_order = action
        elif isinstance(action, CancelOrder):
            #取消订单
            if action.id_to_delete in self.ready_to_execute_orders:
                self.ready_to_execute_orders.pop(action.id_to_delete)
        else:
            assert False, "错误的行动类型!"
```

接下来的 tick 方法用于执行模拟每个 tick 操作,包括更新队列、执行订单和取消订单等,代码如下:

```
#//第 5 章//simulator/simulator.ipynb

class Sim:
    def __init__(…):
        …

    …

    def tick(self) -> Tuple[ float, List[ Union[OwnTrade, MdUpdate] ] ]:
        '''
        模拟一次 tick 操作

        Returns:
            Tuple[ float, List[ Union[OwnTrade, MdUpdate] ] ]:
            - 下一个事件的接收时间戳(receive_ts)或无穷大(np.inf)
```

```
            - 模拟结果,包含 OwnTrade 和 MdUpdate 的列表
        '''
        while True:
            #获取所有队列的事件时间
            strategy_updates_queue_et = self.get_strategy_updates_queue_event_
time()
            md_queue_et = self.get_md_queue_event_time()
            actions_queue_et = self.get_actions_queue_event_time()

            #如果所有队列都为空,则结束模拟
            if md_queue_et == np.inf and actions_queue_et == np.inf:
                break

            #选择具有最小事件时间的队列(strategy queue)
            if strategy_updates_queue_et < min(md_queue_et, actions_queue_et):
                break

            #更新市场数据队列和行动队列
            if md_queue_et <= actions_queue_et:
                self.update_md( self.md_queue.popleft() )
            if actions_queue_et <= md_queue_et:
                self.update_action( self.actions_queue.popleft() )

            #主动执行最后一个订单
            self.execute_last_order()
            #执行具有当前订单簿的订单
            self.execute_orders()
            #删除最后一笔交易
            self.delete_last_trade()

        #模拟结束后处理策略更新队列
        if len(self.strategy_updates_queue) == 0:
            return np.inf, None
        key = list(self.strategy_updates_queue.keys())[0]
        res = self.strategy_updates_queue.pop(key)
        return key, res


    def execute_last_order(self) -> None:
        '''
        尝试主动执行 self.last order 的函数
        '''
        #如果没有要执行的订单,则直接返回
        if self.last_order is None:
            return

        executed_price, execute = None, None
        #如果是买单并且价格大于或等于最佳卖价,则以订单簿价格执行
        if self.last_order.side == 'BID' and self.last_order.price >= self.best_ask:
            executed_price = self.best_ask
            execute = 'BOOK'
        #如果是卖单并且价格小于或等于最佳买价,则以订单簿价格执行
```

```
        elif self.last_order.side == 'ASK' and self.last_order.price <= self.best_bid:
            executed_price = self.best_bid
            execute = 'BOOK'

        #如果执行成功,则创建 OwnTrade 并添加到策略更新队列
        if not executed_price is None:
            executed_order = OwnTrade(
                self.last_order.place_ts, #下单时间
                self.md.exchange_ts, #交易所时间
                self.md.exchange_ts + self.md_latency, #接收时间
                self.get_trade_id(), #交易 ID
                self.last_order.order_id,
                self.last_order.side,
                self.last_order.size,
                executed_price, execute)
            #将订单添加到策略更新队列
            if not executed_order.receive_ts in self.strategy_updates_queue:
                self.strategy_updates_queue[ executed_order.receive_ts ] = []
             self.strategy_updates_queue[ executed_order.receive_ts ].append
(executed_order)
        else:
            #如果执行失败,则将订单添加到等待执行的订单列表
            self.ready_to_execute_orders[self.last_order.order_id] = self.last_order

        #删除最后一个订单
        self.last_order = None


    def execute_orders(self) -> None:
        '''
        执行等待执行的订单
        '''
        executed_orders_id = []
        for order_id, order in self.ready_to_execute_orders.items():

            executed_price, execute = None, None

            #如果是买单并且价格大于或等于最佳卖价,则以订单簿价格执行
            if order.side == 'BID' and order.price >= self.best_ask:
                executed_price = order.price
                execute = 'BOOK'
            #如果是卖单并且价格小于或等于最佳买价,则以订单簿价格执行
            elif order.side == 'ASK' and order.price <= self.best_bid:
                executed_price = order.price
                execute = 'BOOK'
            #如果是买单并且价格大于或等于上一次卖价,则以订单簿价格执行
            elif order.side == 'BID' and order.price >= self.trade_price['ASK']:
                executed_price = order.price
                execute = 'TRADE'
            #如果是卖单并且价格小于或等于上一次买价,则以订单簿价格执行
            elif order.side == 'ASK' and order.price <= self.trade_price['BID']:
```

```
                executed_price = order.price
                execute = 'TRADE'

            #如果执行成功,则创建 OwnTrade 并添加到策略更新队列
            if not executed_price is None:
                executed_order = OwnTrade(
                    order.place_ts, #下单时间
                    self.md.exchange_ts, #交易所时间
                    self.md.exchange_ts + self.md_latency, #接收时间
                    self.get_trade_id(), #交易 ID
                    order_id, order.side, order.size, executed_price, execute)

                executed_orders_id.append(order_id)

                #将订单添加到策略更新队列
                if not executed_order.receive_ts in self.strategy_updates_queue:
                    self.strategy_updates_queue[ executed_order.receive_ts ] = []
                self.strategy_updates_queue[ executed_order.receive_ts ].append
(executed_order)

        #删除已执行的订单
        for k in executed_orders_id:
            self.ready_to_execute_orders.pop(k)


    def place_order(self, ts: float, size: float, side: str, price: float) -> Order:
        '''
        提交订单的函数

        Args:
            ts (float): 下单时间
            size (float): 订单数量
            side (str): 订单方向('BID'或'ASK')
            price (float): 订单价格

        Returns:
            Order: 创建的订单对象
        '''
        #创建订单并添加到行动队列
        order = Order(ts, ts + self.latency, self.get_order_id(), side, size,
price)
        self.actions_queue.append(order)
        return order


    def cancel_order(self, ts: float, id_to_delete: int) -> CancelOrder:
        '''
        取消订单的函数

        Args:
            ts (float): 取消时间
            id_to_delete (int): 要取消的订单 ID
```

```
        Returns:
            CancelOrder: 创建的取消订单对象
        '''
        #创建取消订单并添加到行动队列
        ts += self.latency
        delete_order = CancelOrder(ts, id_to_delete)
        self.actions_queue.append(delete_order)
        return delete_order
```

这样市场状态环境就被 Simulator 模拟呈现了，其中初始化买卖订单中使用了-np.inf 和 np.inf。这个作为启动值是为了方便比较又不失逻辑。为了使时间戳之间的比较顺利地进行，用了 self.get_strategy_updates_queue_event_time()、self.get_md_queue_event_time()、self.get_actions_queue_event_time()方法，最好的做法是将时间戳数值化，这一点在训练开始之前会做相应处理，后面的篇幅中会介绍。这样做的好处是，不让计算烦琐，比起 np.datetime64 和 pandas.datetime64 这样的时间戳类型，数值类型会比较方便。这样也可以和初始值 np.inf 做比较。这个 Sim 类用于模拟交易策略在市场中执行的行为。它接受市场数据作为输入，并根据定义的策略执行交易订单。

接下来介绍./strategies/rl.py 文件中的内容，其中包括 A2CNetwork：一个神经网络模型，用于实现 Actor-Critic 强化学习算法的近似值函数和策略函数，以支持强化学习算法的训练和决策。模型的输入是状态数据，输出是动作概率和值函数的估计。Policy：一个策略对象，该策略对象基于神经网络模型来选择行动。用于强化学习中的策略梯度方法，其中策略由神经网络模型参数化，通过在给定观察值的情况下选择动作，并根据所选行动的性能来更新策略。ComputeValueTargets：用于计算策略梯度方法中的价值目标(Value Targets)，这个类在策略梯度算法中用于计算每个时间步的优势函数估计或用于计算策略梯度损失中的价值目标。这有助于训练 Agent 以改进其策略，从而最大化长期奖励。RLStrategy：一个策略类，这个策略类的主要功能是根据给定的市场数据和策略模型，在模拟交易环境中执行买入和卖出操作，并记录交易和利润等信息。策略的行为根据模型选择的动作来确定，它可以用于测试和训练交易策略，其中库存比率及总盈亏特征将会在这里生成，在 ess 表中游走。A2C：Advantage Actor-Critic 类，一个基于策略梯度的强化学习算法。Evaluate：一个全局函数，用于评估训练好的交易策略在市场环境中的表现。可以在给定市场数据和模拟环境的情况下，对已训练的交易策略进行评估，并获取其在测试数据上的表现结果。这有助于了解策略的泛化能力和实际交易能力。以下是代码部分：

```
#//第 5 章//strategies/rl.ipynb

#导入所需的库和模块
from datetime import timedelta                    #用于处理时间间隔的模块
from typing import List, Optional, Tuple, Union  #用于提供类型提示的模块

import numpy as np                                #用于数值计算和数组操作的 NumPy 库
import pandas as pd                               #用于数据处理和分析的 Pandas 库
import torch                                      #深度学习框架 PyTorch
import torch.nn as nn                             #PyTorch 中的神经网络模块
```

```
import torch.nn.functional as F                    #PyTorch 中的函数模块
from torch.nn.utils import clip_grad_norm_         #用于梯度裁剪的 PyTorch 函数
import wandb                                       #用于实验追踪和记录的 WandB 库

#导入市场环境模拟器相关的类和函数
from simulator. simulator import MdUpdate, Order, OwnTrade, Sim, update _ best _
positions

#evaluate 函数:用于评估策略的性能
def evaluate(strategy, md, latency, md_latency):
    #重置策略的状态
    strategy.reset()
    #创建模拟器,传入市场数据、延迟和市场数据延迟
    sim = Sim(md, latency, md_latency)
    #使用无梯度下降的方式执行策略并获取结果
    with torch.no_grad():
        trades _list, md _list, updates _list, actions _history, trajectory =
strategy.run(sim, mode='test')
    #计算总奖励、策略的总实现盈亏和轨迹
    total_reward = np.sum(trajectory['rewards'])
    total_pnl = strategy.realized_pnl + strategy.unrealized_pnl

    #返回总奖励、总盈亏和轨迹
    return total_reward, total_pnl, trajectory


class A2CNetwork:
  …

class Policy:
  …

class ComputeValueTargets:
  …

class RLStrategy:
  …

class A2C:
  …
```

A2CNetwork 类的代码如下：

```
#//第 5 章//strategies/rl.ipynb

#A2CNetwork 类:表示 Actor-Critic 网络结构
class A2CNetwork(nn.Module):
    '''
    Input:
        状态数据 - 张量,形状为(batch_size x num_features x num_lags)
    output:
        logits - 张量,actor 策略的行动选择概率 logits,形状为(batch _size x num_
actions)
```

```
        V - 张量,critic 的估值,张量,形状为(batch_size)
    '''

    def __init__(self, n_actions, DEVICE="cpu"):
        super().__init__()

        self.DEVICE = DEVICE

        #定义神经网络的结构
        self.backbone = nn.Sequential(
            nn.Flatten(),                          #将输入展平
            nn.Linear(300 *57, 256),
            #线性层,输入维度为 300*57,输出维度为 256
            nn.ReLU()                              #ReLU 激活函数
        )
        self.logits_net = nn.Sequential(
            nn.Linear(256, 128),
             #线性层,输入维度为 256,输出维度为 128
            nn.ReLU(),                             #ReLU 激活函数
            nn.Linear(128, n_actions)
            #线性层,输入维度为 128,输出维度为 n_actions
        )
        self.V_net = nn.Sequential(
            nn.Linear(256, 64),
             #线性层,输入维度为 256,输出维度为 64
            nn.ReLU(),                             #ReLU 激活函数
            nn.Linear(64, 1)                       #线性层,输入维度为 64,输出维度为 1
        )

    #初始化网络权重的函数
    def _init_weights(self, module):
        if isinstance(module, nn.Linear) or isinstance(module, nn.Conv2d):
            #使用正交初始化方法初始化权重
            torch.nn.init.orthogonal_(module.weight.data, np.sqrt(2))
        if module.bias is not None:
            #将偏置项初始化为 0
            module.bias.data.zero_()

    #前向传播函数,接受状态数据作为输入,返回 actor 策略的 logits 和 critic 的估值
    def forward(self, state_t):
        #对输入数据进行前向传播
         hidden_outputs = self.backbone(torch.as_tensor(np.array(state_t),
dtype=torch.float).to(self.DEVICE))
        #返回 actor 策略的 logits 和 critic 的估值,并对 critic 的估值进行挤压(squeeze)
#以匹配形状
         return self.logits_net(hidden_outputs), self.V_net(hidden_outputs).
squeeze()
```

300×57 表示输入特征的总数。这个值表示输入数据的维度,具体来讲,是一个形状为(batch_size, num_features, num_lags)的输入数据的扁平化版本。在神经网络中,输入数据通常被扁平化成一维向量,以便进行全连接层的运算。nn.Linear(300×57, 256)表示一个线性全连接层,它将输入维度为 300×57 的特征转换成维度为 256 的输出特征。这个层

会对输入进行线性变换，其中 300×57 是输入特征的维度，256 是输出特征的维度。输入数据的形状：神经网络的输入是一个三维的数据集，具有以下特性。

batch_size：表示可以同时处理多个时间序列。例如，如果 batch_size 为 32，则在每个训练步骤中会同时处理 32 个不同的时间序列数据。

num_features：这是每个时间步的特征数量。本例中为 57，在 ess 中体现。

num_lags：这是关于时间序列的重要部分。它表示在模型中考虑的历史数据的滞后期数。滞后期数告诉我们，模型会查看多少个先前的时间步进行预测。例如，如果 num_lags 为 10，则模型会考虑过去 10 个时间步的数据来预测未来。

Policy 类的代码如下：

```
#//第 5 章//strategies/rl.ipynb

#Policy 类:表示策略模型
class Policy:
    def __init__(self, model):
        #初始化策略模型
        self.model = model

    #act 方法:执行策略的动作选择
    def act(self, inputs):
        '''
        input:
            inputs - NumPy array, (batch_size x num_features x num_lags)
            输入数据,通常包括批量数据、特征数量和时间滞后(lags)信息
        output: 字典,键名为['actions', 'logits', 'log_probs', 'values']:
            返回一个包含以下键值的字典:
            'actions' - 所选的动作,NumPy 数组,(batch_size)
            'logits' - 动作的 logits,张量,(batch_size x num_actions)
            'log_probs' - 所选动作的 log 概率,张量,(batch_size)
            'values' - critic 的估值,张量,(batch_size)
        '''
        #使用策略模型获取动作的 logits 和评论家的估值
        logits, values = self.model(inputs)

        #使用 Softmax 函数计算动作的概率分布
        probs = F.softmax(logits, dim=-1)

        #随机选择动作,根据概率分布选择动作的索引
        actions = np.array(
            [np.random.choice(a=logits.shape[-1], p=prob, size=1)[0]
             for prob in probs.detach().cpu().NumPy()]
        )

        #添加一个微小的值以避免 log(0)的情况,并计算所选动作的 log 概率
        eps = 1e-7
        log_probs = torch.log(probs + eps)[np.arange(probs.shape[0]), actions]

        #计算动作概率分布的熵
        entropy = -torch.sum(probs *torch.log(probs + eps))
```

```
        #返回动作、logits、log 概率、估值和熵的字典
        return {
            "actions": actions,
            "logits": logits,
            "log_probs": log_probs,
            "values": values,
            "entropy": entropy,
        }
```

ComputeValueTargets 类的代码如下：

```
#//第 5 章//strategies/rl.ipynb

class ComputeValueTargets:
    def __init__(self, policy, gamma=0.999):
        #初始化 ComputeValueTargets 类的实例
        #policy 是一个用于采取动作和计算价值的策略
        #gamma 是折扣因子,用于计算未来奖励的影响程度
        self.policy = policy
        self.gamma = gamma

    def __call__(self, trajectory, latest_observation):
        '''
        此方法应该通过添加一个具有键'value_targets'的项目来修改轨迹(trajectory)

        input:
            trajectory - 来自 runner 的字典,包含有关轨迹的信息
            latest_observation - 最后一种状态观察值,NumPy 数组,形状为(num_envs x
channels x width x height)
        '''
        #为 trajectory 字典添加一个名为'value_targets'的项目,初始化为空列表
        trajectory['value_targets'] = [
            torch.empty(0)
            for _ in range(len(trajectory['values']))
        ]

        #初始化一个列表 value_targets,包含最后一个观察值的策略的估值
        value_targets = [self.policy.act(latest_observation)["values"]]

        #从倒数第 2 个时间步开始,向前计算每个时间步的 value_targets
        for step in range(len(trajectory['values']) - 2, -1, -1):
            value_targets.append(
            #使用折扣因子 gamma 来计算未来奖励的影响,并添加到 value_targets 列表中
                self.gamma *value_targets[-1] + trajectory['rewards'][step]
            )

        #将 value_targets 列表反转,以匹配时间步的顺序
        value_targets.reverse()
        for step in range(len(trajectory['values'])):
        #将计算出的 value_targets 值分配给 trajectory 字典的'value_targets'键
            trajectory['value_targets'][step] = value_targets[step]
```

RLStrategy 类的代码如下：

```
#//第 5 章//strategies/rl.ipynb

class RLStrategy:
    """
    该策略每隔"delay"纳秒发出询价和出价订单
    如果订单在"hold_time"纳秒内未执行,则会被取消
    """

    def __init__(self, policy: Policy, ess_df: pd.DataFrame, max_position:
float,
                 means, stds,
                 delay: float, hold_time: Optional[float] = None, transforms=[],
                 trade_size=0.001, post_only=True, taker_fee=0.0004, maker_fee=
-0.00004) -> None:
        """
        Args:
            policy (Policy): 策略所使用的模型对象
            ess_df (pd.DataFrame): 包含策略特征的数据帧
            max_position (float): 最大持仓量,即策略可以持有的最大资产数量
            means (numpy.ndarray): 特征数据的均值数组
            stds (numpy.ndarray): 特征数据的标准差数组
            delay (float): 下单之间的延迟时间(以纳秒为单位)
            hold_time (Optional[float]): 持仓时间(以纳秒为单位),如果不提供,则默认
为 `delay` 的 5 倍或 10 秒的时间长度
            transforms (list): 用于对轨迹数据进行转换的函数列表
            trade_size (float): 每次交易的资产数量
            post_only (bool): 是否仅使用被动委托(post-only)
            taker_fee (float): 主动吃单(taker)的手续费率
            maker_fee (float): 被动挂单(maker)的手续费率
        """
        self.policy = policy              #设置策略模型
        self.features_df = ess_df         #存储特征数据的数据帧

        #这里将原本 55 个特征值扩增到了 57 个特征值,这两个特征用于 Agent 与
#Environment
        self.features_df['inventory_ratio'] = 0.0    #初始化持仓比率列
        self.features_df['tpnl'] = 0.0               #初始化总收益列

    #num_lags = 300
        self.means = np.broadcast_to(means, (300, 57)).T
#广播均值数组以匹配特征形状
        self.stds = np.broadcast_to(stds, (300, 57)).T
#广播标准差数组以匹配特征形状
        self.max_position = max_position             #最大持仓量
        self.delay = delay                           #下单之间的延迟时间
        if hold_time is None:
#如果未提供持仓时间,则默认为 `delay` 的 5 倍或 10 秒
            hold_time = min(delay * 5, pd.Timedelta(10, 's').delta)
        self.hold_time = hold_time       #持仓时间
        self.coin_position = 0           #初始化资产仓位
        self.realized_pnl = 0            #初始化已实现收益
        self.unrealized_pnl = 0          #初始化未实现收益
        self.action_dict = {1: (0, 0), 2: (0, 4), 3: (0, 9),
```

```
                            4: (4, 0), 5: (4, 4), 6: (4, 9),
                            7: (9, 0), 8: (9, 4), 9: (9, 9)}
                            #定义动作与订单类型的映射字典
        self.actions_history = []           #用于记录历史动作的列表
        self.ongoing_orders = {}            #用于存储未完成订单的字典
        self.trajectory = {}                #存储轨迹数据的字典
        for key in ['actions', 'logits', 'log_probs', 'values', 'entropy',
'observations', 'rewards']:
            self.trajectory[key] = []       #将轨迹数据的各个键值初始化为空列表
        self.transforms = transforms        #轨迹数据转换函数列表
        self.trade_size = trade_size        #每次交易的资产数量
        self.post_only = post_only          #是否仅使用被动委托
        self.taker_fee = taker_fee          #主动吃单(taker)的手续费率
        self.maker_fee = maker_fee          #被动挂单(maker)的手续费率

    def reset(self):
        """
        重置策略状态
        """
        self.features_df['inventory_ratio'] = 0 #重置持仓比率
        self.features_df['tpnl'] = 0            #重置总收益
        self.coin_position = 0                  #重置资产仓位
        self.realized_pnl = 0                   #重置已实现收益
        self.unrealized_pnl = 0                 #重置未实现收益
        self.actions_history = []               #重置动作历史
        self.ongoing_orders = {}                #清空未完成订单
        self.trajectory = {}                    #清空轨迹数据
        for key in ['actions', 'logits', 'log_probs', 'values', 'entropy',
'observations', 'rewards']:
            self.trajectory[key] = []           #重置轨迹数据的各个键值

    def add_ass_features(self, receive_ts) -> None:
        """
        将辅助特征添加到特征数据帧

        Args:
            receive_ts: 接收时间戳
        """
        inventory_ratio = abs(self.coin_position)/self.max_position #计算仓位比率
        tpnl = self.realized_pnl + self.unrealized_pnl #计算总收益
        #更新特征数据帧中对应时间戳的仓位比率和总收益列
        self.features_df.loc[
            self.features_df['receive_ts'] == receive_ts,
            ['inventory_ratio', 'tpnl']
        ] = (inventory_ratio, tpnl)

    def get_features(self, receive_ts):
      #获取特征数据
      features = self.features_df[
          (self.features_df['receive_ts'] <= pd.to_datetime(receive_ts)) &
          (self.features_df['receive_ts'] >= (pd.to_datetime(receive_ts) -
timedelta(seconds=10)))
      ].drop(columns='receive_ts').values.T
```

```
        #如果特征数据的列数小于 300
        if features.shape[1] < 300:
            try:
                #通过填充使用边缘模式来将特征数据的列数扩展到 300
                features = np.pad(features, ((0, 0), (300 - features.shape[1], 0)),
mode='edge')
            except ValueError:
                features = self.means #如果填充失败,则使用预先计算的均值
        #如果特征数据的列数大于 300
        elif features.shape[1] > 300:
            features = features[:, -300:] #仅保留最后的 300 列特征数据

        #对特征数据进行标准化,减去均值并除以标准差
        return (features - self.means) / self.stds

    def place_order(self, sim, action_id, receive_ts, asks, bids):
        if action_id == 0:
            return #如果动作是 0,即无动作,则不执行任何操作

        #如果动作不为 0
        else:
            ask_level, bid_level = self.action_dict[action_id] #获取动作对应的卖方
                                                                #和买方挡位
            ask_order = sim.place_order(receive_ts, self.trade_size, 'ASK', asks
[ask_level]) #下卖单
            bid_order = sim.place_order(receive_ts, self.trade_size, 'BID', bids
[bid_level]) #下买单

            #将下单信息记录到正在进行的订单字典
            self.ongoing_orders[bid_order.order_id] = (bid_order, 'LIMIT')
            self.ongoing_orders[ask_order.order_id] = (ask_order, 'LIMIT')

        #记录动作历史(时间戳、持仓、动作 ID)
        self.actions_history.append((receive_ts, self.coin_position, action_id))


    def run(self, sim: Sim, mode: str, count=1000) -> \
        Tuple[List[OwnTrade], List[MdUpdate], List[Union[OwnTrade, MdUpdate]],
List[Order]]:
        """
        This function runs simulation

        Args:
            sim (Sim): simulator
            mode (str): 运行模式,'train' 或 'test'
            count (int): 运行迭代次数,默认为 1000

        Returns:
            trades_list (List[OwnTrade]): 执行的交易列表
            md_list (List[MdUpdate]): 策略接收的市场数据更新列表
            updates_list (List[Union[OwnTrade, MdUpdate]]): 所有接收的更新列表
            all_orders (List[Order]): 所有下单列表
        """
```

```
        md_list: List[MdUpdate] = []        #用于存储策略接收的市场数据更新列表
        trades_list: List[OwnTrade] = []    #用于存储执行的交易列表
        updates_list = []                   #用于存储所有接收的更新列表
        #当前最佳位置
        best_bid = -np.inf                  #初始化最佳买价
        best_ask = np.inf                   #初始化最佳卖价
        bids = [-np.inf] *10                #初始化 10 个买价
        asks = [np.inf] *10                 #初始化 10 个卖价

        prev_time = -np.inf                 #初始化上一个订单的时间戳
        #尚未执行/取消的订单
        prev_total_pnl = None               #初始化上一个总收益
        if mode != 'train':
            count = 1e8             #如果运行模式不是训练,则将迭代次数设置为一个大的数值
        while len(self.trajectory['rewards']) < count:

            receive_ts, updates = sim.tick()        #从模拟器获取更新
            if updates is None:
                break

            updates_list += updates                 #将更新添加到更新列表中
            for update in updates:

                if isinstance(update, MdUpdate):    #如果更新是市场数据更新
                    if update.orderbook is not None:
                        best_bid, best_ask, asks, bids = update_best_positions(best_
bid, best_ask, update, levels=True)
                    md_list.append(update)
                elif isinstance(update, OwnTrade): #如果更新是自有交易
                    trades_list.append(update)
            #从字典中删除已执行的交易
                    if update.order_id in self.ongoing_orders.keys():
                        _, order_type = self.ongoing_orders[update.order_id]
                        self.ongoing_orders.pop(update.order_id)

                    if self.post_only:
                        if order_type == 'LIMIT' and update.execute == 'TRADE':
                            if update.side == 'BID':
                                self.coin_position += update.size
                                self.realized_pnl -= (1 + self.maker_fee) *update.
price *update.size
                            else:
                                self.coin_position -= update.size
                                self.realized_pnl += (1 - self.maker_fee) *update.
price *update.size
                            self.unrealized_pnl = self.coin_position *((best_ask +
best_bid) / 2)
                        elif order_type == 'MARKET':
                            if update.side == 'BID':
                                self.coin_position += update.size
                                self.realized_pnl -= (1 + self.taker_fee) *update.
price *update.size
                            else:
                                self.coin_position -= update.size
```

```
                            self.realized_pnl += (1 - self.taker_fee) *update.
price *update.size
                        self.unrealized_pnl = self.coin_position *((best_ask +
best_bid) / 2)
                else:
                    if update.execute == 'TRADE':
                        fee = self.maker_fee
                    else:
                        fee = self.taker_fee
                    if update.side == 'BID':
                        self.coin_position += update.size
                        self.realized_pnl -= (1 + fee) *update.price *update.size
                    else:
                        self.coin_position -= update.size
                        self.realized_pnl += (1 - fee) *update.price *update.size
                     self.unrealized_pnl = self.coin_position *((best_ask +
best_bid) / 2)

            else:
                assert False, 'invalid type of update!' #如果更新类型无效,则抛出
#异常
        self.add_ass_features(receive_ts)

        if receive_ts - prev_time >= self.delay:
            if mode == 'train':
                if prev_total_pnl is None:
                    prev_total_pnl = 0
                    prev_coin_pos = 0
                else:
                    if self.coin_position <= 0.2:
                        reward = (
                              self.realized_pnl + self.unrealized_pnl - prev_
total_pnl -
                            0.1 *abs(self.coin_position)
                        )
                    else:
                        reward = -0.2
                    prev_total_pnl = self.realized_pnl + self.unrealized_pnl
                    prev_coin_pos = self.coin_position

                    self.trajectory['observations'].append(features)
                    self.trajectory['rewards'].append(reward)

                    for key, val in act.items():
                        self.trajectory[key].append(val)

            #下单
            features = self.get_features(receive_ts)
            act = self.policy.act([features])
            self.place_order(sim, act['actions'][0], receive_ts, asks, bids)

            prev_time = receive_ts
```

```
            to_cancel = []
            for ID, (order, order_type) in self.ongoing_orders.items():
                if order.place_ts < receive_ts - self.hold_time:
                    sim.cancel_order(receive_ts, ID)
                    to_cancel.append(ID)
            for ID in to_cancel:
                self.ongoing_orders.pop(ID)

        if mode == 'train':
            for transform in self.transforms:
                transform(self.trajectory, [features])

        return trades_list, md_list, updates_list, self.actions_history,
self.trajectory
```

A2C 类的代码如下：

```
#//第 5 章//strategies/rl.ipynb

class A2C:
    def __init__(self, policy, optimizer, value_loss_coef=0.1, entropy_coef=0.1,
max_grad_norm=0.5, DEVICE="cpu"):
        """
        初始化 A2C(Advantage Actor-Critic)算法的训练器

        Args:
            policy: 策略网络,用于生成动作
            optimizer: 优化器,用于更新策略网络的参数
            value_loss_coef: 价值损失系数,控制价值损失在总损失中的权重
            entropy_coef: 熵损失系数,控制熵损失在总损失中的权重
            max_grad_norm: 梯度裁剪的阈值,用于稳定训练过程
            DEVICE: 设备(例如 "cpu" 或 "cuda")

        Attributes:
            policy: 策略网络
            optimizer: 优化器
            value_loss_coef: 价值损失系数
            entropy_coef: 熵损失系数
            max_grad_norm: 梯度裁剪的阈值
            DEVICE: 训练所使用的设备
            last_trajectories: 最后一次训练的轨迹数据
        """
        self.policy = policy
        self.optimizer = optimizer
        self.value_loss_coef = value_loss_coef
        self.entropy_coef = entropy_coef
        self.max_grad_norm = max_grad_norm
        self.DEVICE = DEVICE

        self.last_trajectories = None      #用于存储最后一次训练的轨迹数据

    def loss(self, trajectory):
        """
        计算 A2C 算法的损失函数
```

```
        Args:
            trajectory: 轨迹数据,包含动作概率、估计值、实际值等信息

        Returns:
            total_loss: 总损失,包括策略损失、价值损失和熵损失的组合

        注意:在计算损失时,应注意使用适当的权重来调整价值损失和熵损失的重要性

        这种方法的注释描述了如何计算 A2C 算法的损失函数,包括策略损失、价值损失和熵损失
的权衡
        """
        #将轨迹中的数据堆叠为 PyTorch 张量,并将其移到指定的设备上
        trajectory['log_probs'] = torch.stack(trajectory['log_probs']).squeeze().
to(self.DEVICE)
        trajectory['value_targets'] = torch.stack(trajectory['value_targets']).to
(self.DEVICE)
         trajectory['values'] = torch.stack(trajectory['values']).to(self.
DEVICE)
        trajectory['entropy'] = torch.stack(trajectory['entropy']).to(self.
DEVICE)

        #计算策略损失、价值损失和熵损失
        policy_loss = (trajectory['log_probs'] *(trajectory['value_targets'] -
trajectory['values']).detach()).mean()
         critic_loss = ((trajectory['value_targets'].detach() - trajectory
['values']) **2).mean()
        entropy_loss = trajectory["entropy"].mean()

        #计算总损失,由策略损失、价值损失和熵损失组成,并使用权重调整它们
         total_loss = self.value_loss_coef *critic_loss - policy_loss - self.
entropy_coef *entropy_loss

        #记录各种损失和训练奖励
        wandb.log({
            'total loss': total_loss.detach().item(),
            'policy loss': policy_loss.detach().item(),
            'critic loss': critic_loss.detach().item(),
            'entropy loss': entropy_loss.detach().item(),
            'train reward': np.mean(trajectory['rewards'])
        })

        return total_loss
```

5.10.4　模型训练

一切开始之前,载入必要的模块,代码如下:

```
#//第 5 章//training.ipynb

#用于序列化和反序列化 Python 对象
import pickle
#用于生成随机数
```

```
import random
#用于科学计算的库
import numpy as np
#用于数据处理的库
import pandas as pd
#PyTorch 深度学习库
import torch

#以下是自定义的模块
#Sim 是用于模拟市场环境的类
from simulator.simulator import Sim
#包含了神经网络结构、策略迭代、强化学习策略、A2C 算法等类
from strategies. rl import A2CNetwork, Policy, RLStrategy, A2C,
ComputeValueTargets, evaluate
#整理数据类型作为模型输入,思路是把 trades 数据做成 AnonTrade 对象、把 lob 数据做成
#OrderbookSnapshotUpdate 对象(一个瞬间的订单快照),最终把 AnonTrade 及
#OrderbookSnapshotUpdate 合成为 MdUpdate 对象,作为一个市场瞬间数据,使用多个
#MdUpdate 对象组成一个列表,成为一个市场时间序列数据
from simulator.simulator import AnonTrade, MdUpdate, OrderbookSnapshotUpdate
```

数据准备,代码如下:

```
#//第 5 章//rl_training.ipynb

lobs = './data/orderbook.csv'
trades = './data/trade.csv'

#设置随机值生成参数,使随机数据可以复现
seed = 13
random.seed(seed)
np.random.seed(seed)
torch.manual_seed(seed)

#设置 GPU 或 CPU
DEVICE = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

生成 OrderbookSnapshotUpdate 对象,代码如下:

```
#//第 5 章//rl_training.ipynb

lobs = pd.read_csv(lobs)
lobs['receive_ts'] = pd.to_datetime(lobs['receive_ts'], errors='coerce')
lobs['exchange_ts'] = pd.to_datetime(lobs['exchange_ts'], errors='coerce')
receive_ts = lobs.receive_ts.values
exchange_ts = lobs.exchange_ts.values

asks = [list(zip(lobs[f"ask_price_{i}"], lobs[f"ask_vol_{i}"])) for i in range
(10)]
asks = [[asks[i][j] for i in range(len(asks))] for j in range(len(asks[0]))]
bids = [list(zip(lobs[f"bid_price_{i}"], lobs[f"bid_vol_{i}"])) for i in range
(10)]
bids = [[bids[i][j] for i in range(len(bids))] for j in range(len(bids[0]))]

#books 列表包含多个 OrderbookSnapshotUpdate 对象,每个对象代表一个订单簿的快照更新
books = list (OrderbookSnapshotUpdate (* args) for args in zip (exchange _ts,
receive_ts, asks, bids))
```

生成 AnonTrade 对象，代码如下：

```
#//第 5 章//rl_training.ipynb

trades = pd.read_csv(trades)
trades['aggro_side'] = trades['aggro_side'].astype(str)
trades['receive_ts'] = pd.to_datetime(trades['receive_ts'], errors='coerce')
trades['exchange_ts'] = pd.to_datetime(trades['exchange_ts'], errors='coerce')

receive_ts = trades.receive_ts.values
exchange_ts = trades.exchange_ts.values

trades = trades[['exchange_ts', 'receive_ts', 'aggro_side', 'size', 'price']].
sort_values(["exchange_ts", 'receive_ts'])

#trades 列表包含多个 AnonTrade 对象，每个对象代表一笔匿名交易
trades = [AnonTrade(*args) for args in trades.values]
```

时间戳格式转换，代码如下：

```
#//第 5 章//rl_training.ipynb

#将所有时间戳内容转换为 1970 年 1 月 1 日之后的纳秒数
for i in range(len(trades)):
  trades[i].exchange_ts = (trades[i].exchange_ts - np.datetime64('1970-01-
01T00:00:00')) //np.timedelta64(1, 'ns')
  trades[i].receive_ts = (trades[i].receive_ts - np.datetime64('1970-01-01T00:
00:00')) //np.timedelta64(1, 'ns')

for i in range(len(books)):
  books[i].exchange_ts = (books[i].exchange_ts - np.datetime64('1970-01-01T00:
00:00')) //np.timedelta64(1, 'ns')
  books[i].receive_ts = (books[i].receive_ts - np.datetime64('1970-01-01T00:00:
00')) //np.timedelta64(1, 'ns')
```

合成 MdUpdate 对象，代码如下：

```
#//第 5 章//rl_training.ipynb

"""
  创建一个名为 trades_dict 的字典，用于将交易数据组织成键-值对的形式
  键是由 trade 对象的 exchange_ts 和 receive_ts 组成的元组
  值是对应的 trade 对象本身
"""
trades_dict = {(trade.exchange_ts, trade.receive_ts): trade for trade in trades}

"""
  创建一个名为 books_dict 的字典，用于将订单簿数据组织成键-值对的形式
  键是由 book 对象的 exchange_ts 和 receive_ts 组成的元组
  值是对应的 book 对象本身
"""
books_dict = {(book.exchange_ts, book.receive_ts): book for book in books}

#以上两行代码的主要目的是将成交数据和订单簿数据从列表结构转换为字典结构，以便能够根据
#成交数据或订单簿的时间戳快速检索和访问相关数据
```

```
#字典的键是时间戳的组合,而值是对应的数据对象,这种组织方式通常有助于提高数据的检索
#效率

#ts,时间序列合并
ts = sorted((key for key in (trades_dict.keys() | books_dict.keys()) if not any
(pd.isna(k) for k in key)))

#生成一个包含多个 MdUpdate 对象的列表,其中每个对象都对应于 ts 列表中的一个时间戳,并包
#含与该时间戳相关的 books_dict 和 trades_dict 中的数据(如果存在)
md = [MdUpdate(*key, books_dict.get(key, None), trades_dict.get(key, None)) for
key in ts]
```

在 md 这个列表中,每个元素都是一个 MdUpdate 对象,只要任意一个元素满足,要么 orderbook 有内容,要么 trade 有内容,无内容为 None,代码如下:

```
#//第 5 章//rl_training.ipynb
[MdUpdate(exchange_ts=1694508640976000000, receive_ts=1694508640992000000,
orderbook=OrderbookSnapshotUpdate(exchange_ts=1694508640976000000, receive_ts
=1694508640992000000, asks=[(ap0, av0),…,(ap9,av9)], bids=[(bp0, bv0),…,(bp9,
bv9)]), trade=None),
…,
MdUpdate(exchange_ts=1655942409193000000, receive_ts=1655942409197011118,
orderbook=None, trade=AnonTrade(exchange_ts=1655942409193000000, receive_ts=
1655942409197011118, side='BID', size=…, price=…)),
…]
```

载入特征数据,代码如下:

```
#//第 5 章//rl_training.ipynb

with open('./data/features_dict.pickle', 'rb') as f:
  ess_dict = pickle.load(f)

ess_df = pd.DataFrame.from_dict(ess_dict, orient='index').reset_index().rename
(columns={'index': 'receive_ts'})

with open('./data/means.npy', 'rb') as f:
  means = np.load(f, allow_pickle=True)

with open('./data/stds.npy', 'rb') as f:
  stds = np.load(f, allow_pickle=True)
```

划分训练集测试集,代码如下:

```
#//第 5 章//rl_training.ipynb

train_len = math.floor(len(md) *.8)
test_len = len(md) - train_len
md_train = md[:train_len]
md_test = md[train_len:len(md)]
```

实例化一个 A2CNetwork 对象作为强化学习策略的神经网络模型。这个模型被设计用于处理具有 10 个不同行动的任务，然后使用创建的神经网络模型创建一个 Policy 对象，该对象将使用该模型来执行动作选择，代码如下：

```
#//第 5 章//rl_training.ipynb

#神经网络模型可以在指定的计算设备(DEVICE)上进行计算
model = A2CNetwork(n_actions=10, DEVICE=DEVICE).to(DEVICE)
policy = Policy(model)
```

神经网络结构，代码如下：

```
#//第 5 章//rl_training.ipynb

A2CNetwork(
  (backbone): Sequential(
    (0): Flatten(start_dim=1, end_dim=-1)
    (1): Linear(in_features=17100, out_features=256, bias=True)
    (2): ReLU()
  )
  (logits_net): Sequential(
    (0): Linear(in_features=256, out_features=128, bias=True)
    (1): ReLU()
    (2): Linear(in_features=128, out_features=10, bias=True)
    )
  (V_net): Sequential(
    (0): Linear(in_features=256, out_features=64, bias=True)
    (1): ReLU()
    (2): Linear(in_features=64, out_features=1, bias=True)
  )
)
```

初始化一个用于交易策略的对象 strategy，优化器 optimizer，以及一个基于 A2C 算法的强化学习 Agent，代码如下：

```
#//第 5 章//rl_training.ipynb

#创建一个延迟(delay)为 0.1s 和持有时间(hold_time)为 10s 的策略，以用于交易策略的定义
#这个策略将使用指定的神经网络策略(policy)来决定交易行为，其中包括动作选择和订单执行
#配置交易价值目标计算(ComputeValueTargets)，以及一些交易参数、费率参数等参数
delay = pd.Timedelta(0.1, 's').total_seconds()
hold_time = pd.Timedelta(10, 's').total_seconds()
strategy = RLStrategy(policy, ess_df, 1.0, means, stds, delay, hold_time,
[ComputeValueTargets(policy)],
                      trade_size=0.01, post_only=True, taker_fee=0.0004, maker_
fee=-0.00004)

#创建一个用于强化学习训练的优化器(optimizer)
#这个优化器将用于更新神经网络模型的权重，以最小化损失函数，从而改进策略的性能
optimizer = torch.optim.RMSprop(model.parameters(), lr=7e-4, alpha=0.99, eps=1e-5)

#创建一个基于 Advantage Actor-Critic(A2C)算法的强化学习 Agent(a2c)
```

```
#这个 Agent 将使用上述定义的策略(policy)和优化器(optimizer)来执行训练
#它还配置了值函数损失的权重(value_loss_coef)和熵损失的权重(entropy_coef)
#可以指定计算设备(DEVICE),用于神经网络计算
a2c = A2C(policy, optimizer, value_loss_coef=0.25, entropy_coef=1, DEVICE=
DEVICE)
```

模型训练,代码如下:

```
#//第 5 章//rl_training.ipynb

import wandb
import os

#设置 WANDB_PYTHON_NAME 环境变量
os.environ['WANDB_PYTHON_NAME'] = 'training.py'
wandb.login()
wandb.init(project="MarketMaking")
#监控模型的性能,以便实时可视化和记录模型的训练进展
wandb.watch(model)

#导入 tqdm 库的 trange 函数,用于显示训练循环的进度条
from tqdm.Notebook import trange

#使用 trange 创建一个从 1 到 5000 的循环,每个迭代代表一个训练周期(epoch)
for i in trange(1, 5001):
    print(f'epoch {i}')

    #调用 A2C 代理的 train 方法来执行训练
    #这里训练了策略(strategy),并使用了一些训练数据(md_train)及其他参数
    a2c.train(strategy, md_train,
              latency=pd.Timedelta(10, 'ms').total_seconds(),
              md_latency=pd.Timedelta(10, 'ms').total_seconds(),
              count=290,
              train_slice=600_000)

    #如果当前训练周期是 500 的倍数,则对模型性能进行评估
    if i % 500 == 0:
        #调用 evaluate 函数来评估策略在测试数据集上的性能
        reward, pnl, trajectory = evaluate(strategy,
                                   md_test,
                                   latency=pd.Timedelta(10, 'ms').total_seconds(),
                                   md_latency=pd.Timedelta(10, 'ms').total_seconds())

        #使用 wandb.log 记录评估结果,这将有助于监控策略的性能
        wandb.log({
            'val reward': reward,
            'val pnl': pnl,
        })
```

效果如图 5-32 所示。

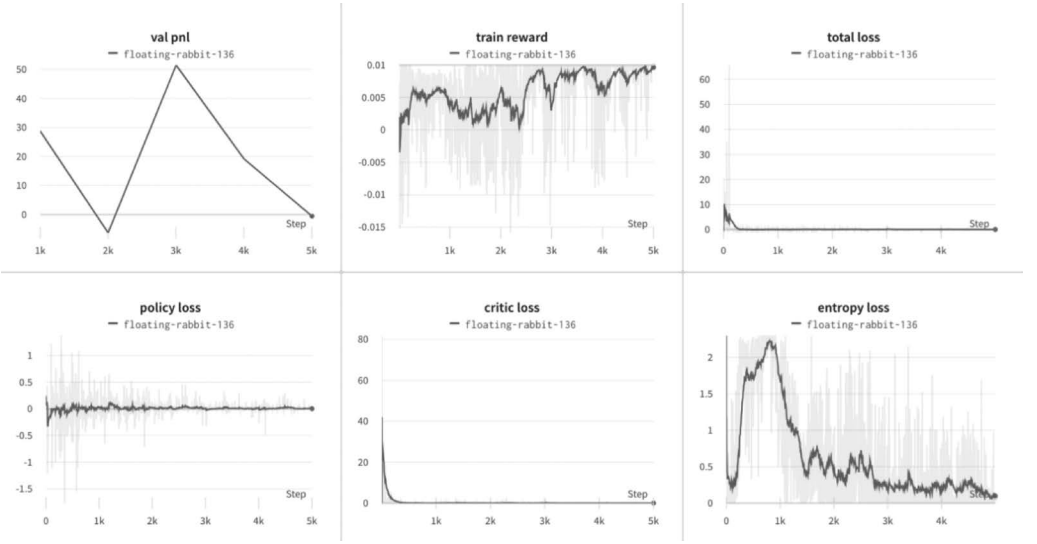

图 5-32　模型效果图

总体来讲，这个模型的环境状态空间有中间价距离、累计名义价值、市场不平衡度、订单流平衡度、相对强弱指数、价差等特征。Agent 状态空间有库存比率、总盈亏特征。行动状态空间见表 5-3。

表 5-3　行动状态空间

ActionID	1	2	3	4	5	6	7	8	9	0
Ask	1	1	1	5	5	5	10	10	10	—
Bid	1	5	10	1	5	10	1	5	10	—

带库存惩罚的 PnL 计算：

$$r_t = \Delta q_t (p_t - c_t) + \mathrm{PNL}_t^{\mathrm{realized}} - \alpha |q_t| \tag{5-212}$$

其中，r_t 表示在 t 时刻的回报，Δq_t 表示在 t 时刻的持仓变化，c_t 表示在 t 时刻的平均成本，α 表示库存惩罚系数，$|q_t|$ 表示 t 时刻的库存绝对值，$\mathrm{PNL}_t^{\mathrm{realized}}$ 表示 t 时刻的已实现盈亏。

A2C 更新公式（策略梯度）：

$$\nabla_\theta J(\theta) = \nabla_\theta \ln \pi(a_t \mid s_t; \theta') A(s_t, a_t; \theta, \theta_v) \tag{5-213}$$

其中，$\nabla_\theta J(\theta)$ 表示目标函数 $J(\theta)$ 关于策略参数 θ 的梯度；这个梯度用于更新策略参数，以最大化总体回报。

$\nabla_\theta \ln \pi(a_t | s_t; \theta')$：表示在给定策略参数 θ' 下，行动 a_t 在状态 s_t 下的对数概率的梯度；这个概率通常由策略网络（Policy Network）计算。

$A(s_t, a_t; \theta, \theta_v)$：表示优势函数（Advantage Function），它用于衡量在状态 s_t 下采取行动 a_t 相对于平均水平的优势。它的计算方式通常是：$A(s_t, a_t; \theta, \theta_v) = Q(s_t, a_t; \theta_v) - V(s_t; \theta_v)$，其中 $Q(s_t, a_t; \theta_v)$ 是状态动作对 (s_t, a_t) 的估值，$V(s_t; \theta_v)$ 是状态 s_t 的估值。

θ 是策略函数（Actor）的参数，用于定义在给定状态下选择行动的概率分布。

θ 是值函数(Critic)的参数,用于定义值函数,该值函数用于估计状态的价值或回报,帮助策略函数更好地选择行动。

函数逼近,如图 5-33 所示。

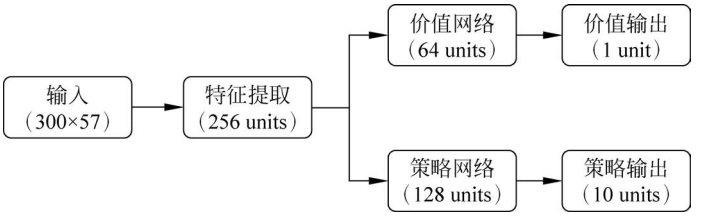

图 5-33 函数逼近

这样的思路,可以对高频数据进行广泛实验,以证明强化学习方法的有效性,同时也可以确定其局限性。强化学习方法的局限性包括需要算法训练(需要大量的时间和计算资源)、大量影响最终结果并需要调整的超参数,以及强化学习策略的推理速度。